Change distilling. Enriching software evolution analysis with fine-grained source code change histories

Abstract

Software systems have to evolve over their life-cycle or they become progressively less useful. The reasons of why software is continuously changed are manifold: Features are added or adapted because of changing requirements; bugs have to be fixed because of faults in the software; or the software has to be migrated because of modernization. One negative effect of the continuing change is the software aging phenomenon. As software is changed from people unaware of the initial design concepts and, mostly, under time-pressure software becomes larger, more complex, and less understandable. As a result, in the last decade, several techniques have been developed to understand the negative impact of continuing change by analyzing change in general and source code change in particular.

The approaches developed so far suffer from the coarse-grained information available for changes. They rely on data provided by versioning systems, which keep track of changes by storing the text differences of a particular file. Changes at the level of source code entities are not considered. In addition, a precise definition and a classification of source code changes are still missing. Both are key to extract and analyze source code changes, and eventually understand the negative impact of continuing change. We therefore claim: Extracting, classifying, and analyzing finegrained source code changes from the history of software systems provide useful insights into problems of continuing change and can identify support mechanisms to reduce them.

The key contribution of this dissertation is change distilling, a methodology to define, classify, extract, and analyze fine-grained source code changes. Change distilling provides a taxonomy of source code changes which defines source code change types according to tree edit operations in the abstract syntax tree. Our change distilling algorithm applies tree differencing pairwise on subsequent versions of abstract syntax trees to extract the tree edit operations.

We provide three empirical experiments to show the benefits of extracting finegrained source code change types. First, we analyze the source code and comment co-change behavior in the evolution of eight software systems. We show that in cases where comments are adapted to source code changes, the related changes happen in the same revision. We also show that in half of these software systems API comments are adapted several revisions after the source code change happened.

Second, we explore whether certain change types appear frequently together. For that we use hierarchical agglomerative clustering to discover change type patterns and present a catalogue of change type patterns. The results from a commercial software system show that certain control flow changes are due to source code cleanup activities, that exception flow is used differently in different system parts, and that API convention changes are spread over many releases.

Third, we investigate whether methods exist whose invocations are significantly more affected by context and update changes than other methods, and whether we can reveal change patterns among these invocation changes. We develop an approach that ranks how often context and update changes were applied to invocations of a particular method and whether these changes were bug fixes. In addition, we extract patterns of context and update changes to assess whether they can be used to provide valuable change suggestions.

The results of our three software evolution experiments provide enough evidence that the analysis of

change types helps in understanding software evolution and provides means to support developers in their daily work.

Beat Fluri

Change Distilling

Enriching Software Evolution Analysis with Fine-Grained Source Code Change Histories

Dissertation

Advisor
Prof. Dr. Harald C. Gall
University of Zurich

External Examiner
Prof. Dr. David Notkin
University of Washington

s.e.a.l. – Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

http://seal.ifi.uzh.ch/fluri
http://fluri.computerscience.ch

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Change Distilling

Enriching Software Evolution Analysis with Fine-Grained Source Code Change Histories

Dissertation

for the Degree of a
Doctor in Informatics

at the Faculty of Economics,
Business Administration and
Information Technology
of the
University of Zurich

by
Beat Fluri

from
Basel, BS, Switzerland

Accepted on the recommendation of
Prof. Dr. Harald C. Gall
Prof. Dr. David Notkin

October 2008

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, October 22, 2008

The Vice Dean of the Academic Program in Informatics: Prof. Dr. Harald C. Gall

Acknowledgements

Many people accompanied me during the endeavor of my doctoral studies and of writing my dissertation. I am grateful for their support.

First of all, I would like to thank my advisor, Harald Gall, for showing me the joy of research, for his unending encouragement when I was in doubt about my work, for his professional guidance, for teaching me how to write, for creating an environment to enjoy work and research, and for becoming a friend.

Special thanks go to David Notkin for enthusiastically accepting to be my external examiner, for reviewing my dissertation, for his professional advices, and for the fruitful discussions we have when we meet.

I thank Michele Lanza for reviewing my dissertation, for teaching me that passion is the key to success, and for the cool time at various ICSEs (especially in Minneapolis).

Many thanks goes to Martin Pinzger with whom I shared the office during my doctoral studies. Martin—after telling me which office space I have to take—became a good friend. I thank Martin for all the interesting technical discussion, for his honest criticism, for teaching me how to write papers, and for all the fun during our "non-technical" conversations. Martin, I really enjoyed the time we had.

Thanks to my fellow doctoral students, particularly to Michael Würsch and Emanuel Giger for their valuable comments and suggestions, and for the intensive but interesting time during paper writing. To Geri Reif, Patrick Knab, and all other s.e.a.l. members. To Christoph Kiefer for the good time we had during school, our studies at ETH, and our doctoral studies. To Esther Kaufmann for her help and for the fun we had in the D-Dock. To Avi Bernstein for his advices.

Thanks to all the diploma students for their important contributions: Edoardo Beutler, Reto Geiger, Emanuel Giger, Marco Jakob, Michael Würsch, and Jonas Zuberbühler.

I thank my best friend, Philipp Traber, very much for supporting and encouraging me in whatever I do, for pushing me to write a dissertation, for always being there when I need him, and for the wonderful time we always have. To my dear friend Patrice Mercier for all the fun we have and for always surprising me anew. To Christian Döbeli for his advice.

I thank my family-in-law for making me feel part of them.

I am grateful to my parents, Sylvia and Peter, for their love, for their unconditional support, and for believing in me—whenever and wherever. To my brother, Christoph with Stefanie.

My deepest and sincere thanks go to my wife. I am thankful for her constant support, for her endless patience when I am obsessed with my passions, for always believing in me, and for her love. Janine, I am grateful to have you. This work is dedicated to you.

—BEAT FLURI

University of Zurich, Switzerland
October 2008

To Janine

Abstract

Software systems have to evolve over their life-cycle or they become progressively less useful. The reasons of why software is continuously changed are manifold: Features are added or adapted because of changing requirements; bugs have to be fixed because of faults in the software; or the software has to be migrated because of modernization. One negative effect of the continuing change is the *software aging* phenomenon. As software is changed from people unaware of the initial design concepts and, mostly, under time-pressure software becomes larger, more complex, and less understandable. As a result, in the last decade, several techniques have been developed to understand the negative impact of continuing change by analyzing *change* in general and *source code change* in particular.

The approaches developed so far suffer from the coarse-grained information available for changes. They rely on data provided by versioning systems, which keep track of changes by storing the text differences of a particular file. Changes at the level of source code entities are not considered. In addition, a precise definition and a classification of source code changes are still missing. Both are key to extract and analyze source code changes, and eventually understand the negative impact of continuing change. We therefore claim: *Extracting, classifying, and analyzing fine-grained source code changes from the history of software systems provide useful insights into problems of continuing change and can identify support mechanisms to reduce them.*

The key contribution of this dissertation is *change distilling*, a methodology to define, classify, extract, and analyze fine-grained source code changes. Change distilling provides a *taxonomy of source code changes* which defines source code *change types* according to tree edit operations in the abstract syntax tree. Our change distilling algorithm applies tree differencing pairwise on subsequent versions of abstract syntax trees to extract the tree edit operations.

We provide three empirical experiments to show the benefits of extracting fine-grained source code change types. First, we analyze the source code and comment co-change behavior in the evolution of eight software systems. We show that in cases where comments are adapted to source code changes, the related changes happen in the same revision. We also show that in half of these software systems API comments are adapted several revisions after the source code change happened.

Second, we explore whether certain change types appear frequently together. For that we use hierarchical agglomerative clustering to discover change type patterns and present a catalogue of change type patterns. The results from a commercial software system show that certain control flow changes are due to source code cleanup activities, that exception flow is used differently in different system parts, and that API convention changes are spread over many releases.

Third, we investigate whether methods exist whose invocations are significantly more affected by context and update changes than other methods, and whether we can reveal change patterns among these invocation changes. We develop an approach that ranks how often context and update changes were applied to invocations of a particular method and whether these changes were bug fixes. In addition, we extract patterns of context and update changes to assess whether they can be used to provide valuable change suggestions.

The results of our three software evolution experiments provide enough evidence that the analysis of change types helps in understanding software evolution and provides means to support developers in their daily work.

Zusammenfassung

Software Systeme müssen sich während ihres Lebenszyklus weiterentwickeln oder sie werden stufenweise unbrauchbarer. Die Gründe warum Software sich kontinuierlich ändert sind mannigfaltig: Funktionalität wird ergänzt; Fehler müssen behoben werden; oder die Software muss wegen Modernisierung migriert werden. Ein negativer Effekt dieser kontinuierlichen Änderung ist das Phänomen der *Software Alterung*. Da Software von Leuten, die vom anfängliche Entwurf der Software keine Kenntnis haben und meistens unter Zeitdruck arbeiten müssen, geändert wird, vergrössert sich die Software, wird komplizierter und weniger verständlich. Aus diesem Grund wurden im letzten Jahrzehnt verschiedene Techniken zum Verständnis der negativen Auswirkungen der kontinuierlichen Änderung entwickelt. Diese Techniken analysieren *Änderungen* im Allgemeinen und *Programmcode-Änderungen* im Speziellen.

Die bis anhin entwickelten Ansätze leiden an den zur Verfügung stehenden, grobkörnigen Änderungsinformationen aus Versionierungssystemen. Diese verfolgen Änderungen indem sie Unterschiede auf der Ebene des Textes, jedoch nicht der Programmcode-Entitäten, einer Datei speichern. Zudem fehlen eine klare Definition von Programmcode-Änderungen und deren Klassifizierung. Beide sind zur Extraktion und Analyse von Programmcode-Änderungen äusserst wichtig, um letztendlich die negativen Auswirkungen der kontinuierlichen Änderung zu verstehen. Wir behaupten deshalb: *Die Extraktion, Klassifikation und Analyse von feinkörnigen Programmcode-Änderungen aus der Geschichte eines Software Systems liefern nützliche Erkenntnisse über die Probleme der kontinuierlichen Änderung und können Hilfestellungs-Mechanismen identifizieren, um diese zu vermindern.*

Der Hauptbeitrag dieser Dissertation ist "*change distilling*," eine Methodik zur Definition, Klassifikation, Extraktion und Analyse von feinkörnigen Programmcode-Änderungen. "Change distilling" liefert eine *Taxonomie von Programmcode-Änderungen*, welche *Programmcode-Änderungsarten* anhand von Baum-Editier-Operationen im abstrakten Syntax-Baum definiert. Um solche Operationen zu extrahieren, wendet unser "change distilling"-Algorithmus Baum-Vergleiche paarweise an aufeinanderfolgende Versionen eines abstrakten Syntax-Baums an.

Um den Nutzen der Extraktion von feinkörnigen Programmcode-Änderungsarten zu zeigen, beschreiben wir drei empirische Experimente. Erstens analysieren

wir in der Evolution von acht Software Systemen das Verhalten von gemeinsamen Änderungen zwischen Programmcode und Kommentaren. Wir zeigen, dass wenn Kommentare an Programmcode-Änderungen angepasst werden, diese Änderungen in der gleichen Version passieren. Wir zeigen zudem, dass in der Hälfte dieser Systeme API-Kommentare erst mehrere Versionen später an die Programmcode-Änderungen angepasst werden.

Zweitens erkunden wir ob gewisse Änderungsarten oft miteinander auftreten. Wir verwenden "hierarchisches agglomeratives Clustering," um Muster unter Änderungsarten zu erkennen und präsentieren die gefundenen in einem Katalog. Die Resultate aus einer Studie mit einem kommerziellen Software System zeigen, dass gewisse Kontrolfluss-Änderungen auf das Programmcode-Aufräumen zurück zu führen sind, dass Programm-Ausnahmeverhalten unterschiedlich in verschiedenen Systemteilen angewendet werden und dass Änderungen von API-Konventionen über mehrere Versionen verteilt sind.

Drittens untersuchen wir ob Methoden existieren, deren Aufrufe von Kontext- und Aktualisierungs-Änderungen signifikant stärker betroffen sind als andere und ob wir Muster zwischen diesen Aufruf-Änderungen finden können. Wir entwickeln einen Ansatz, der Methoden anhand der Häufigkeit und Fehlerbehebungs-Charakter von Kontext- und Aktualisierungs-Änderungen von deren Aufrufen ordnen. Zusätzlich extrahieren wir Muster zwischen diesen Änderungen, um zu beurteilen, ob diese nützliche Änderungsvorschläge liefern können.

Die Resultate unserer drei Software Evolution Experimente liefern genügend Nachweis, dass die Analyse von Änderungsarten zum Verständnis der Software Evolution beiträgt und Mittel zur Unterstützung von Software Entwicklern bereitstellt.

Contents

List of Figures

List of Tables

Those who cannot remember the past are condemned to repeat it.

—George Santayana (1863–1952)
The Life of Reason, Volume 1, 1905

I

Setting the Scene

1 Introduction

MAINTENANCE is the most time and resource consuming task in the lifecycle of a software system: In the early eighties the cost for software maintenance was estimated at 40 to 70 percent of the total development costs (Boehm, 1981; Pfleeger and Atlee, 2006). Nowadays, estimates suggest that maintenance costs have increased to 80 percent (Pfleeger and Atlee, 2006) or even to 90 percent (Erlikh, 2000).

The first reason why software has to be maintained is manifested in Lehman's (1980) *Laws of Program Evolution*. According to Lehman, software has to undergo continual change or it becomes progressively less useful. Many software systems represent business processes which have to be adapted continuously; either because of changing environments, of business reorientation, or of modernization. The corresponding software systems have then to be adapted to these external influences.

Second, many software bugs are observed and reported when a software system is in operation. The bugs are then corrected by changing the software.

Third, a software system is never completed. Several requirements do not arise until users experience what features their new software could additionally provide (Brooks, 1995). That leads to the integration of further features after a software system is released.

One negative effect of this continuing change is the *software aging* phenomenon (Parnas, 1994): "Programs, like people, get old." Software ages because of the changes that are made to it. Parnas calls this effect *ignorant surgery*. That means, software is changed from different people who are not aware of the software and especially not of its intended design concepts. Moreover, customers and managers

demand that bugs are fixed quickly. This time pressure inhibits developers to understand the software and its design concepts to their full extent before they fix bugs. As a consequence, changes will be inconsistent and the software becomes eventually larger, more complex, and less understandable. Ignorant surgery is therefore the primary cause of the maintenance cost explosion: As the complexity impacts understandability, increasing complexity implies increasing maintenance effort. We might also speak about the vicious circle in software maintenance.

To understand why software systems become less maintainable when changed continuously, and to reduce their maintenance costs eventually, we have to investigate their change histories. The research field of this investigation is known as *software evolution analysis*. It is the retrospective analysis of the evolution, *i.e.*, history, of a software system. The evolution comprises all activities in the life-cycle of a software system, beginning with requirements analysis until its shutdown. According to Mens and Demeyer (2008, Chp. 1) analyzing historical data of a software system has two dimensions. Both dimensions aim at limiting the effect of software aging and reducing the maintenance cost. They are defined as follows:

1. *What and why*. The what and why dimension focuses on the *nature* of the software evolution phenomenon. For instance, it tries to answer why continuing change increases the complexity of a software system. We call this dimension *understanding* throughout the dissertation.

2. *How*. The how dimension focuses on aiding developers or project managers in their daily business. For instance, with recommender systems that are configured with software evolution data we can provide feedback during development. We call this dimension *support* throughout the dissertation.

With the approaches and investigations presented in this dissertation, we contribute to both dimensions.

1.1 Motivation and Thesis

Since Lehman's Laws of Program Evolution from the 1980s, it is well understood that software has to be adapted to changing requirements and environments or it becomes progressively less useful. Change is broadly accepted as a crucial part of a software's life-cycle. Because of the law of *Continuing Change* software ages and source code tends to decay (Eick *et al.*, 2001): "Code is decayed if it is *harder to change than it should be*." Code decay does not mean that software is inherently hard to change because of its essential complexity. It rather means that source code becomes more difficult to change than it used to be because continuing change increases its complexity. Code decays when fixing bugs or adding new features implies increasing changes (perception by developers), increasing need for resources

(perception by managers), and increasing costs (perception by customers). According to Eick *et al.*, code decay is indicated by source code that is frequently changed as well as frequently fixed, or by widely dispersed source code changes.

As a result, in the last decade, several techniques have been developed to understand the impact of continuing change to code decay. For instance, Gall *et al.* (1998) detected possible maintainability hot-spots by analyzing co-change relationships of modules that point to hidden dependencies and shortcomings of the structure of a software system. Ying *et al.* (2004) and Zimmermann *et al.* (2005) developed approaches that guide programmers along related changes by telling them "programmers who changed these functions also changed..." In contrast to the work of Gall *et al.*, they used changes on the method level instead of the file level. The *Hipikat* tool of Čubranić *et al.* (2005) used project history information to provide recommendations for a modification task. These approaches started with putting *change* in general and *source code change* in particular in the center of the software evolution analysis.

We argue that such techniques and tools are valuable but suffer from the coarse-grained information available for changes. They rely on data provided by versioning systems, which keep track of changes by storing the text differences of a particular file. Structural changes in the source code are not considered. We have shown that by using structural source code changes we can already filter a high amount (about 50 percent) of co-change relationships that did not have any source code changes (Fluri *et al.*, 2005).

Therefore, an approach to extract more detailed change information is necessary to better understand the negative consequences of continuing change. Partially this has been realized by several approaches (see Chapter 2 for a discussion), but either they still rely on textual differences enhanced with source code information or extract particular kinds of changes for specific tasks. These approaches are able to narrow down changes to the method level but fail in further qualifying changes, such as, the addition of a method invocation in the else-part of an if-statement.

The notion of change in general and of source code change in particular was used at different levels of granularity in several software evolution studies. In early change history investigations a source code change was defined as a change in a source file regardless what entities inside the file changed (Fischer *et al.*, 2003b; Gall *et al.*, 1998; Ying *et al.*, 2004). The next level of granularity was to define a change at the level of source code lines. A change was either a change of a single source code line (Purushothaman and Perry, 2005; Zimmermann *et al.*, 2005) or a change of a block of lines (Kim *et al.*, 2006a). The analysis of source code changes at the AST level or the control flow graph (CFG) has increased its influence in recent years (Apiwattanapong *et al.*, 2007; Raghavan *et al.*, 2004). To our knowledge the most fine-grained level is recording every key stroke of developers (Robbes and Lanza, 2007a).

These different definitions hinder the discussion, extraction, and analysis of source code changes. Therefore, we need to define source code changes precisely and at different levels of granularity. By accumulating changes over time we can then (1) identify patterns of source code changes to highlight similar change activities; or (2) build change histories for different types of source code entities (*e.g.*, classes or methods) to highlight similarly changed entities.

Furthermore, a classification of changes according to their impact on other source code entities and whether they are functionality-preserving or functionality-modifying does not exist yet. This information is important to distinguish significant from irrelevant changes which improves the quality of software evolution analysis results. For instance, we can filter evolutionary hotspots as found by Gall *et al.* (1998) and provide only those hotspots that change significantly.

We claim that defining, classifying, and extracting fine-grained source code changes is important to improving the quality of software evolution analysis results and, as a consequence, to providing better support for programmers. For instance, the *Hatari* tool rates the risk of changing a method according to the frequency of method changes that caused a bug (Śliwerski *et al.*, 2005a). Detailed information about the changes is not taken into account, *e.g.*, whether a bug is caused by the insertion of a method invocation statement or by the insertion of a whole else-if-statement. With the fine-grained change information such a differentiation would be possible: *Hatari* could inform software developers which source code changes in which parts of the method body are risky to apply.

To summarize, we state

> **Thesis**:
> *Extracting, classifying, and analyzing fine-grained source code changes from the history of software systems provide useful insights into problems of continuing change and can identify support mechanisms to reduce them.*

1.2 Change Distilling in a Nutshell

In this dissertation we propose *change distilling* to define, classify, and analyze fine-grained source code changes. Change distilling provides a *taxonomy of source code changes* that precisely defines *change types* according to tree edit operations in the abstract syntax tree (AST). We use the basic tree edit operations *insert*, *delete*, *move*, and *update* applicable to AST nodes. In addition, the taxonomy classifies each change type according to a *change significance level* scheme. The change significance level expresses the possible impact a change type may have on other source code entities and whether it may be functionality-preserving or functionality-modifying.

As a result, the taxonomy defines 40 change types for source code entity types

that are defined in a programming language and representable in an AST. The change types are divided into *body* and *declaration* part categories of attributes, classes, and methods. Each change type obtains a change significance level of *none*, *low*, *medium*, *high*, or *crucial*. For certain change types the change significance level is adapted to the accessibility modifier of a source code entity. For instance, a return type change of a public method has a higher change significance level than of a private method.

We use the taxonomy to extract fine-grained source code changes. Our change distilling algorithm uses tree differencing on subsequent AST versions of an object-oriented class. The algorithm calculates an *edit script* that contains basic tree edit operations and transforms the older into the newer AST. That means, the edit script contains exactly those source code changes that were applied to the class between the two versions. The change distilling algorithm is based on the tree differencing algorithm presented by Chawathe *et al.* (1996) that we customize to be applicable on pairs of ASTs. Our CHANGEDISTILLER is the implementation of the change distilling algorithm for the Java programming language and is integrated into the Eclipse IDE (des Rivières and Wiegand, 2004) as a plugin.

Leveraging the information provided by ASTs permits us to get precise information about a source code change. In addition to the information that a particular source code entity has changed, tree edit operations also provide information about the location of the change. For instance, we can tell that the method invocation statement `foo.bar()` was moved from the then-part to the else-part of the if-statement that has the condition `foo == null`.

To provide evidence that analyzing change types in the history of a software system contributes to the understanding and support of software evolution we present three different *change type-based software evolution analyzes*.

1.3 Thesis Statement

To verify our thesis we rely on the fulfillment of research goals. In this section we first formulate the underlying hypotheses for the research goals and then discuss the goals.

1.3.1 Research Hypotheses

For the fulfillment of our research goals presented in the next section we rely on the acceptance of the following hypotheses.

Change type definition hypothesis

HYPOTHESIS (H1a): *Tree edit operations that correctly transform an abstract syntax tree allow for a precise definition of change types and permit their extraction with tree differencing.*

Source code is hierarchically structured text. Compilers construct a tree representation of the source code to translate the text into machine readable commands. The tree representation is also known as *abstract syntax tree* (AST). Tree edit operations that correctly transform an AST can be leveraged to describe change types in source code. By applying tree differencing on pairs of ASTs we can extract change types automatically.

We verify this hypothesis with respect to object-oriented programming languages using Java as the exemplar. In addition, we present our change distilling algorithm with its implementation, the CHANGEDISTILLER, to extract change types from the history of Java software systems.

Change type classification hypothesis

HYPOTHESIS (H1b): *The change type indicates the impact that it may have on source code entities and whether it may be functionality-preserving or functionality-modifying.*

We assume that the definition of change types also enables their classification. The classification should express their possible change impact on other source code entities. Moreover, the classification should also include whether a change type may be functionality-preserving or functionality-modifying. On purpose we use the term *may* because the examination of a single source code change cannot always reveal its full impact on the software system.

We introduce a *change significance level* scheme that classifies each change type with respect to this hypothesis and argue with examples for its acceptance. Such an argumentation is valid because the acceptance of any other hypothesis and the fulfillment of the research goals do not directly depend on the acceptance of Hypothesis H1b.

Co-change hypothesis

HYPOTHESIS (H2a): *The analysis of comments and source code entities that changed together enables the reasoning about the commenting process of developers in a software system.*

Comments describe single or blocks of source code entities. If this description does not conform to the source code anymore, comments should be adapted. By

associating comments with source code entities we assume that related changes, *i.e.*, co-changes, of comments and source code indicate whether the comment is kept up-to-date or re-documentation took place.

Change activity hypothesis

HYPOTHESIS (H2b): *The analysis of sets of change types that appear frequently together over time allows for the categorization of change and development activities during the evolution of a software system.*

Development activities range from feature implementation or bug fixing to restructuring or cleanup changes. We show that *change type patterns*, *i.e.*, change types that appear frequently together over time, highlight certain development activities.

Method invocation change hypothesis

HYPOTHESIS (H2c): *The analysis of method invocation changes enables the ranking of methods whose invocations are affected by context changes as well as update changes, and reveals patterns among these changes.*

Method invocations are the most used source code entity types—at least in the software systems we investigated. Hence, they are among the most changed entities. This is not surprising because in the object-oriented programming paradigm messages between objects are exchanged by calling methods. We suppose that invocations of particular methods tend to change more often than of other methods and are also more often involved in bug fixes. Therefore, the types and frequency of method invocation changes indicate which methods developers should use with care. Furthermore, we assume that changes on invocations of a method are similar. We can therefore support developers with change suggestions while they add method invocations.

1.3.2 Research Goals

For the success of this dissertation we shall fulfill the following research goals.

Taxonomy of source code changes

RESEARCH GOAL (G1): *Create a taxonomy of source code changes that defines and classifies source code change types precisely.*

The taxonomy relies on the Hypotheses H1a and H1b. The change types that the taxonomy defines must be commonly understandable but have to be precisely defined, too. The classification reflects the impact a change type may have on other

source code entities and whether it may be functionality-modifying or functionality-preserving.

Source code change extraction algorithm and implementation

RESEARCH GOAL (G2): *Define an algorithm and its implementation to extract source code changes from historical data of a software system.*

The algorithm should use the taxonomy that is provided by accepting Hypothesis H1a as well as H1b, and by fulfilling Research Goal G1. Therefore, the algorithm relies on tree differencing pairwise on ASTs. The implementation is integrated into a development environment, such as Eclipse, to enable the extraction of source code changes from the historical data of a software system.

Change type based software evolution analysis

RESEARCH GOAL (G3): *Provide empirical evidence that by analyzing change types we contribute to the understanding of software evolution and identify ways to provide suggestive feedback during development.*

This research goal is closely related to our thesis. Its success depends on Hypotheses H2a–H2c. That means, whether or not we are able to give evidence that change type-based software evolution analysis contributes to the *understanding* as well as to the *support* of software evolution. Hence, we do not only show that we can learn from analyzing source code changes but also that we can provide suggestive feedback during development.

1.3.3 Interrelationship of Research Hypotheses and Goals

In this dissertation we aim at verifying whether change distilling contributes to the understanding of software evolution, and identifies ways to support software evolution. To show how the hypotheses and research goals are connected to our thesis we present their interrelationship in Figure 1.1. The colored rectangles indicate the relationship between the hypotheses and the research goals.

Hypothesis H1a represents the foundation of our thesis and, therefore, our starting point. Rejecting H1a implies that change types cannot be defined. As a consequence, neither can change types be classified, be extracted, nor can we reason about their meaning in software evolution. Rejecting H1a also means that we are not able to develop and implement an algorithm that extracts source code changes in the AST. Thus, we would not be able to analyze source code changes applied during the evolution of a software system and can neither contribute to the *understanding* nor to the *support* of software evolution. Reasons for rejecting

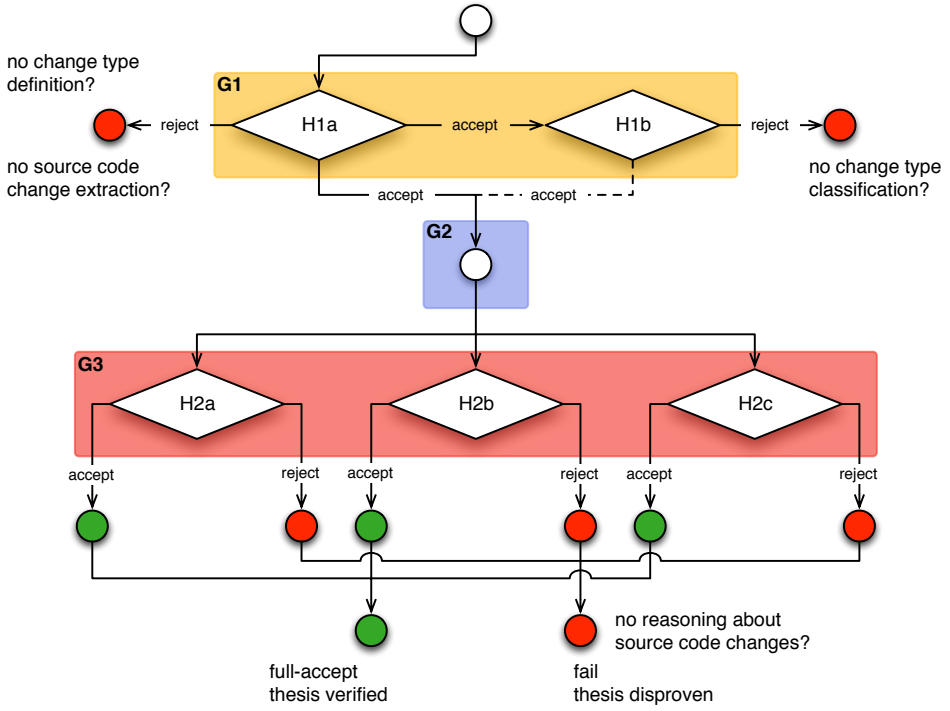


Figure 1.1: Interrelationship of research hypotheses and goals

Hypothesis H1a could be that tree differencing does not provide accurate results, that it does not scale, or that versions of software systems are seldom compilable—mandatory for generating an AST.

The Hypothesis H1b does not impact the acceptance of H2a–H2c. Thus [H1a $\wedge$ $\neg$H1b] is acceptable for the outcome of the dissertation.

The Hypotheses H2a–H2c are independent from each other but contribute to our third research goal and to the verification of our thesis. For the verification of our thesis we demand the acceptance of two out of the three H2 hypotheses. For instance, [H2a $\wedge$ $\neg$H2b $\wedge$ H2c] is acceptable. Reasons for rejecting any of these hypotheses could be that software systems do not have a certain change process, that development activities are in such a way mixed that a focus cannot be recognized, or that the changes on method invocations do not reveal patterns for change suggestions.

A success of our dissertation, in particular of our change distilling approach, is accepting H1a and any two of H2a–H2c.

1.4 Foundation of the Dissertation

The foundation of the dissertation is a set of selected publications. Their relation and their arrangement in the dissertation are depicted in Figure 1.2.

The main contributions in the course of defining a taxonomy of source code changes and a corresponding extraction algorithm are summarized as follows:

1. We described an initial study to find out whether or not it is worth to extract changes at a finer-grained level than the file level (Fluri *et al.*, 2005).
2. We presented the taxonomy of source code changes (Fluri and Gall, 2006).
3. Our change distilling algorithm is the core contribution of this dissertation. We presented tree differencing for fine-grained source code change extraction and the validation of the change distilling algorithm (Fluri *et al.*, 2007a).

Approaches and studies that apply change distilling for software evolution analysis build on top of the first set of publications:

1. We described an initial study of the co-evolution of comments and source code (Fluri *et al.*, 2007b). We then improved the approach and defined hypotheses to statistically test the research questions addressed in the initial study. We also extended the corpus of the experiments from three to eight software systems (Fluri *et al.*, 2008a).
2. We presented an approach to extract and analyze change type patterns (Fluri *et al.*, 2008b).
3. We investigated whether we can leverage context and update changes of method invocations to facilitate the use of methods (Fluri and Gall, 2008). We also described the approach to rank methods whose invocations are affected by context and update changes.

1.5 Contributions

The conceptual contributions of this dissertation focus on the approaches to define and classify source code changes, to extract fine-grained source code changes with tree differencing, and enriching software evolution analysis with change histories. The technical contributions of this dissertation concentrate on the development of tools to extract source code changes, *e.g.*, CHANGEDISTILLER, and conducting the analysis of source code changes. The empirical contributions of this dissertation are the applications of the proposed approaches and techniques to the evolution of open-source and commercial software systems.

The main contributions of this dissertation are:

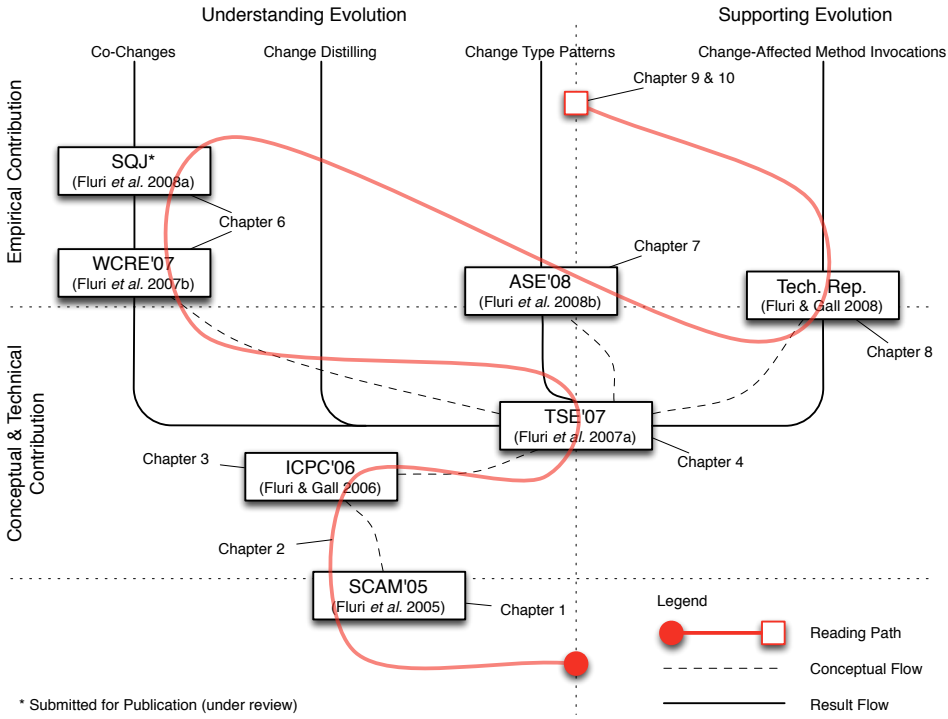


Figure 1.2: Logical relation (bottom-up) of publications that build the foundation of this dissertation

1. A *taxonomy of source code changes* that defines and classifies change types according to tree edit operations in the abstract syntax tree.

2. The *change distilling* algorithm to extract fine-grained source code changes based on tree differencing in the abstract syntax tree.

3. CHANGEDISTILLER, a tool integrated in Eclipse as a plugin to extract fine-grained source code changes from historical data of Java software systems.

4. An approach to associate comments to source code changes. The approach permits us to empirically analyze the co-evolution of comments and source code changes.

5. An approach to discover change type patterns. The approach is based on agglomerative hierarchical clustering of change types that frequently occur together in the course of the evolution of a software system. The approach permits us to categorize development activities.

6. An approach to rank methods whose invocations are affected by context and update changes and to extract patterns among these invocation changes. The approach permits us to facilitate the use of methods by providing suggestive feedback during development.

1.6 Roadmap

The remainder of this dissertation provides the following content:

Chapter 2 (p.17) reviews related work in the field of software evolution in general, source code change extraction, source code change analysis, and clustering of software artifacts.

Chapter 3 (p.31) presents our *taxonomy of source code changes*. We show the definition of change types with basic tree edit operations and how we assign a *change significance level* to each change type.

Chapter 4 (p.51) describes our *change distilling algorithm* to extract change types between subsequent versions of classes. We describe how we customized the original tree differencing algorithm by Chawathe *et al.* (1996) to be applicable to pairs of ASTs. We also present our *benchmark* to validate our algorithm. The benchmark consists of 1,064 manually classified changes in 219 revision of eight different method histories from three different open-source software projects. The algorithm is able to extract source code changes with a mean absolute percentage error of 34 percent; 45 percent better than the original algorithm.

Chapter 5 (p.83) presents CHANGEDISTILLER, our implementation of the change distilling algorithm. CHANGEDISTILLER is integrated into Eclipse as a plug-in. To extract the changes it obtains subsequent revisions of Java classes, applies the change distilling algorithm, and stores instances of our *change history meta model*. The following chapters show several approaches to analyze software evolution based on change types. They all leverage data extracted by CHANGEDISTILLER.

Chapter 6 (p.95) examines the questions whether developers comment their code and to which extent they add comments or adapt them when they evolve the code. We present an approach to associate comments with source code entities to observe their co-evolution over multiple versions. We use a set of heuristics to decide whether a comment is associated to its preceding or its succeeding source code entity. For the observation of the interaction between comments and their associated source code entities we conduct an empirical experiment with eight software system from different domains.

Chapter 7 (p.125) explores whether change types appear frequently together over time. We investigate whether they describe specific development activities such as cleanups, paradigm shifts, or flow alterations. For that a semi-automated approach to discover change type patterns using agglomerative hierarchical clustering is described. Our approach can discover those control flow changes that are due to particular source code cleanup activities, whether exception flow is used differently in different system parts, and whether API convention changes are spread over many releases.

Chapter 8 (p.147) investigates whether methods exist whose invocations are significantly more affected by context and update changes than other methods and whether we can reveal change patterns among these invocation changes. We develop an approach that ranks how often context and update changes were applied to invocations of a particular method and whether these changes were bug fixes. In addition, we extract patterns of context and update changes to assess whether they can be used to provide valuable change suggestions. We perform experiments on three core components of Eclipse: Core, JDT, and PDE. We apply our approach to rank the methods called in Eclipse and extract change patterns.

Chapter 9 (p.175) discusses our contributions to software engineering by elaborating to what extent this dissertation permits us to contribute to the *understanding* and to the *support* of software evolution.

Chapter 10 (p.187) presents conclusions and outlines future work.

Appendix A (p.195) presents the complete list of defined change types.

Appendix B (p.199) presents the selected set of method histories that we used for our benchmark.

Appendix C (p.201) presents the *versioning meta model* used by our software evolution analysis platform, the EVOLIZER.

Appendix D (p.203) presents further results of the co-evolution investigation of comments and source code.

Appendix E (p.207) presents all dendrograms with highlighted change types clusters that we discuss in Chapter 7.

Appendix F (p.211) presents further results of the method ranking and change pattern extraction.

Appendix G (p.217) presents selected publications.

2 Related Work

SOFTWARE ENGINEERING is concerned with the phenomenon of software evolution since software was developed. For instance, the well-known *waterfall life-cycle* process of Royce (1970) describes a naive view on software evolution. However, the term *software evolution* was not deliberately used until Lehman (1980) introduced the *Laws of Program Evolution*. According to Mens and Demeyer (2008, Chp. 1) it took another ten years until the term gained acceptance in the software engineering research community: More appropriate software life-cycle processes, such as the *spiral model* of Boehm (1988), have not become popular until the early nineties.

Software evolution analysis is an active field of research nowadays because the access to historical data is provided by software repositories. Since such repositories are also used in the open-source community, software evolution researchers can obtain a huge amount of software evolution data in different domains. Software evolution analysis by means of software repositories is also called *mining software repositories*. The approaches presented in this dissertation also use software repositories to verify the hypotheses and fulfill the research goals.

In this chapter we present the state-of-the-art of source code change analysis approaches that are related to this dissertation. We start by giving an overview on mining software repositories (Section 2.1) and review techniques to extract source code changes (Section 2.2). We also report on work that present the application of change analysis to support software evolution (Sections 2.3–2.4) and on cluster analysis in software engineering (Section 2.5). We conclude this chapter with a summary of the strengths of change distilling compared to existing approaches (Section 2.6). We refer to (Madhavji *et al.*, 2006) as well as (Mens and Demeyer,

2008) for a more general view on software evolution.

2.1 Mining Software Repositories

The amount of work related to mining software repositories grew over the last decade. Instead of giving a survey on this area we present the approaches that have been driving our research.

Gall *et al.* (1998) were among the first to analyze co-changes in software systems. They introduced the notion of *logical couplings*. Their idea was to detect hidden dependencies between modules by analyzing co-change relationships between files. These hidden dependencies are not evident in source code and, therefore, cannot be detected with static analysis. Fischer *et al.* (2003b) extended the concept of logical coupling and defined a filtering mechanism as well as a data scheme for such an integration. The data scheme was the initial version of the *release history database* (RHDB). The RHDB approach was further adapted for the *ArchView* approach (Pinzger *et al.*, 2005a,b) to identify structural and evolutionary shortcomings in software systems.

By instrumenting the code and analyzing bug reports Fischer *et al.* (2003a) showed how features are scattered over the project tree and how features are logically coupled over releases. An extension of this approach with a number of specific visualization techniques is described in (Fischer and Gall, 2004) and (Fischer, 2006). The extension allows an engineer to uncover hidden dependencies among different features over many releases.

Ying *et al.* (2004) and Zimmermann *et al.* (2005) developed approaches that guide programmers along related changes by telling them "programmers who changed these functions also changed..." In contrast to the work of Gall *et al.* they used changes at the method level instead of the file level. The *Hipikat* tool of Čubranić *et al.* (2005) uses project history information to provide recommendations for a modification task. These approaches started with putting *change* in general and *source code change* in particular in the center of the software evolution analysis. Similar in spirit to these works is the study of DeLine *et al.* (2005). Their *wear-based filtering* approach captures the program interaction history of developers familiar with the system to support developers unfamiliar with the system.

The design of our change history meta model relies on *Hismo* (Gîrba, 2005; Gîrba and Ducasse, 2006), a generic model that treats history as a first class entity. *Hismo* was also used in relation of source code changes: Gîrba *et al.* (2005) used the lines added/deleted information provided by CVS to show how the code ownership of a particular file changes.

A taxonomy of approaches to analyze source code repositories for understanding software evolution was given by Kagdi *et al.* (2005).

2.2 Source Code Change Extraction

Source code differencing has proven itself to be a long-term research topic fundamental to multi-version program analyzes, as pointed out by Kim and Notkin (2006). Existing approaches either rely on lexical, syntactical, or semantical differencing techniques. A further classification can be done with respect to the granularity of the algorithms, *i.e.*, whether they perform coarse-grained or fine-grained change extraction as well as analyzes. Our change distilling algorithm identifies fine-grained syntactical changes. Hassan and Holt (2004) proposed evolutionary code extractors in general. They discussed the need of such tools as well as the level of source code extraction granularity.

In the remainder of this section we first justify the use of Chawathe *et al.*'s tree differencing algorithm as the basis for our change distilling algorithm. Then, we present the three differencing techniques to extract source code changes: (1) textual differencing, (2) syntactic differencing, and (3) semantic differencing. We close this section with related work in the area of refactoring detection, code clone detection, and source code merging.

Tree differencing. The algorithm presented by Chawathe *et al.* (1996) as well as our change distilling algorithm are closely related to tree differencing in general and to the tree edit distance problem in particular. The goal of tree edit distance problem is "to compute the edit distance based on a corresponding edit script between two labeled ordered or unordered trees" (Bille, 2005). The edit operations used are: (1) change the label of a node (relabel), (2) delete a non-root node, and 3) insert a node. One of the first non-naive algorithms for the tree edit distance problem was introduced by Tai (1979). The quadratic upper bound of this general approach was improved by Shasha and Zhang (1990) and Zhang (1995). These algorithms are inappropriate for our concerns: (1) they do not act on labeled, ordered, and valued trees, as it is the case of ASTs, (2) the operation *relabel* cannot be used for source code, as, for instance, a method invocation must not become an assignment, (3) they do not support move operations, and (4) do not support updates of values. The algorithm of Chawathe *et al.* addresses these issues and, additionally, is faster than these general tree edit distance algorithms.

Textual differencing. Existing differencing tools such as the well-known *GNU diff* (Hunt and McIlroy, 1976) deal with flat, rather than with hierarchical information. They are usually based on the *longest common subsequence* (LCS) algorithm (Hunt and Szymanski, 1977) and calculate textual changes, *i.e.*, a list of lines that were changed, inserted, or deleted. *GNU diff* cannot, for example, distinguish between changes applied to license information or documentation and changes applied to a method body. In contrast to *diff*, our change distilling algorithm can detect changes

more precisely and is able to assign a particular change to a concrete source code entity (such as the declaration or body part of a method), rather than just to a line number.

Maletic and Collard (2004) presented a language independent approach for detecting syntactic differences between source files using an intermediate representation of the source code in XML. The output provided by *GNU diff* is mapped to the XML representation to locate changed entities. Our approach does not rely on textual differences and is able to detect changes due to move operations.

Canfora *et al.* (2007) reconstructed changes from differencing results provided by CVS or Subversion *diff* to track the evolution of source code lines. For that, their algorithm uses Vector Space Models and the Levenshtein string similarity measure. But, it cannot detect move changes.

Syntactic differencing. Yang (1991) described an algorithm based on a branch-and-bound implementation of the largest common subtree problem. The output of the algorithm are sets of matching and modified abstract syntax tree nodes, but he did not reporte which operations transform the original into the modified tree.

Raghavan *et al.* (2004) implemented *Dex*, a tool for extracting changes between C source files. They used change information provided by patch files to locate the changed parts in source files. These parts were fed into their tree differencing algorithm that outputs the edit operations. *Dex* also operates on the AST and can be used with our taxonomy to classify source code changes in C programs.

To the best of our knowledge Robbes and Lanza (2007a) extracted changes on the most fine-grained level. They presented and approach to enable fine-grained version control. Their *SpyWare* tool tracks every key-stroke that developers make during development (Robbes and Lanza, 2008). With the recorded change data they can track developer activities, accumulate changes to different levels of granularity, and describe development session (Robbes and Lanza, 2007b).

Semantic differencing. Horwitz's (1990) approach computes semantic as well as textual differences between two programs. The approach partitions a program according to its behavior extracted from the program representation graph. Similar to change distilling, Horwitz built a matching set between such partitions to extract the differences. The approach is limited to programs written in a language that supports a subset of traditional programming languages. Furthermore, our approach provides a more complete set of tree edit operations and additionally classifies changes into change types.

The algorithm presented by Jackson and Ladd (1994) reports semantic changes in procedural programs. They analyzed the input-output behavior of two procedures to detect changed behavior.

Apiwattanapong *et al.* (2007) used enhanced control-flow graphs to model semantic behavior of methods of object-oriented programs. Identifying modified

and unmodified methods was based on graph isomorphism. Their discussion of the impact of path changes caused by exception handling can be used to extend our work. Furthermore, we claim that both approaches, the one presented by Apiwattanapong *et al.* and our work, are complementary and that semantic differencing can be used to extend and refine our work.

Refactoring detection. Although our change distilling algorithm does not aim at detecting refactorings, approaches in extracting them are related to source code change extraction in general. Weissgerber and Diehl (2006) used lightweight regular expressions to extract source code changes for identifying refactorings in CVS repositories. The generated refactoring candidates were ranked using code clone detection. With this approach they reach a high accuracy.

RefactoringCrawler (Dig *et al.*, 2006) takes two lightweight ASTs and uses *Shingles*, a custom syntactic similarity analysis tool, to find similar pairs of source code entities. With semantic analysis of the calculated pairs they detect common refactorings.

Xing and Stoulia (2005b) presented their *UMLDiff* tool, which tracks changes on the interface (logical design) of classes. In contrast to our work, they are able to track when entities are moved among different classes. However, *UMLDiff* focuses on *recovering higher-level design knowledge evolution*, *i.e.*, changes on the interface level, whereas our work additionally allows for fine-grained differencing on the implementation-level. Similar to *UMLDiff*, *SiDiff* by Kelter *et al.* (2005) extracts differences between UML models. The models are stored in XMI files. They used a combined top-down and bottom-up approach for matching model parts. The matching is then used to classify differences of UML models.

Kim *et al.* (2007a) presented an approach to automatically infer likely changes at or above the method level. They used a simple matching algorithm with the Levenshtein string similarity measure.

Code clone detection. The area of code clone detection, although not directly related to our work, rely on source code differencing. Sager *et al.* (2006) used several tree matching algorithms for detecting similar Java classes. First, they converted the abstract syntax tree as generated by Eclipse into the language independent meta model FAMIX (Demeyer *et al.*, 2001). In a second step, they transformed the model into a generic tree format. The generic tree representations of all classes of a software system were then matched against each other to find similar classes. Sager *et al.* evaluated three different tree similarity algorithms for this purpose, derived from a *bottom-up maximum common subtree isomorphism*, a *top-down maximum common subtree isomorphism*, and an *edit distance of two given trees*, all three originally presented in (Valiente, 2002). These algorithms can be used to replace the tree similarity measure calculated in our approach.

Baxter *et al.* (1998) described *CloneDr*, a tool for code clone detection that relies on abstract syntax trees, but categorizes subtrees by hashing. This significantly reduces the number of comparisons needed because only subtrees with the same hash values have to be compared. Classification using hash values works well for exact duplicates, but fails for locating near-miss clones, *i.e.*, code duplicates that are very similar. They were able to overcome this shortcoming by choosing an artificial bad hash function, *i.e.*, a function that ignores identifier names. For determining the similarity of two ASTs, Baxter *et al.* used the Dice Coefficient (Dice, 1945).

Tu and Godfrey (2002) used their *BEAGLE* tool to detect structural evolution of software systems. With origin analysis *BEAGLE* detects old functions as the "origin" of new ones based on software metrics and code clone detection. Origin analysis was also used to detect merging and splitting (Godfrey and Zou, 2005) and method renaming (Kim *et al.*, 2005).

Source code merging. The area of merging also relies on source code differencing. Mens (2002) has conducted a survey on existing software merging techniques. For example, the approaches presented in Asklund (1994); Hunt (2001); Westfechtel (1991); Yang (1994) rely on tree-based differencing to perform merging. All of them have some limitations in terms of our concerns; as far as we know, neither of them detects *moves* or outputs an edit-script.

2.3 Source Code Change Analysis

Our change type-based software evolution experiments are related with source code change analysis. Next we discuss work that has been done in investigating single changes and patterns of changes.

Change in general. Xing and Stoulia (2005a) used their *UMLDiff* to classify interface changes. For each class version they assigned a volatility level, *e.g.*, "intense evolution" or "rapidly developing," according to the number of changes occurred. In contrast, Kelter *et al.* (2005) focused on special differences of UML models (*e.g.*, attribute or reference differences) instead of general insert, delete, move, and update of UML diagram parts. Compared to their work, we classify individual changes.

The *Hatari* tool (Śliwerski *et al.*, 2005a) rates the risk of changing a method according to the frequency of method changes that caused a bug. Such changes are also called fix-inducing changes, meaning that a bug was reported after a particular change was made. The concepts of fix-inducing changes and their identification were described by Śliwerski *et al.* (2005b) and Kim *et al.* (2006a). Small changes were also investigated by Purushothaman and Perry (2005). In a large case study,

they found that there is less than a four percent probability that a one-line change will introduce a fault.

Comment-related analyzes. Jiang and Hassan (2006) conducted a study on the evolution of comments in PostgreSQL. They investigated how many header comments and non-header comments were added or removed to PostgreSQL over time. In contrast to their work, we do not restrict ourselves on studying the addition and deletion of comments, but also track updates and moves. Moreover, we integrate source code change analysis down to the statement level to track whether and how source code and comments change together.

Antoniol *et al.* (2002) proposed an approach based on information retrieval to recover traceability links between source code and free text documents. Marcus and Maletic (2003) proposed a similar solution. However, both approaches focus on external documentation and do not investigate evolutionary aspects, *i.e.*, they do not track documentation and source code changes together over time. Information retrieval techniques were also employed by Lawrie *et al.* (2006) to measure how the comments relate to the source code and assume that comments impact the code quality of software systems. Marcus and Poshyvanyk (2005) defined metrics for measuring the conceptual cohesion of classes. For that, they incorporated the presence (absence) of comments.

Recently Witte *et al.* (2007) used Semantic Web Technologies to connect software and documentation artefacts. They developed ontologies to query the linking.

Ying *et al.* (2005) investigated the usage of a particular type of comment, the Eclipse task comments, *i.e.*, special comments starting with `//TODO` which are common used by developers using the Eclipse IDE. They argued that task comments tend to depend a lot on the context of the surrounding code and that it is difficult to infer the scope of a task comment. This often holds for comments in general and has therefore an impact on our work. Ying *et al.* mentioned a few reasons that lead to an insert of a comment task (for example as pointers to change requests) but they did not study whether some building blocks of a program (*e.g.*, if-statement) are more likely to be commented. Again, they did not analyze any evolutionary aspects, neither of source code nor comment.

Schreck *et al.* (2007) analyzed the quality evolution of comments in the Eclipse project. They used several metrics, such as completeness and quantity, of the comments. With their approach, they found, for instance, strong jumps in the documentation quality of Eclipse—an indication for re-documentation.

Whether false comments may have any impact on bugs was analyzed by Tan *et al.* (2007). They extracted implicit program rules out of comments to automatically detect inconsistencies between comments and source code. For that, they used natural language processing and machine learning. With this approach, Tan *et al.* found new bugs in several open-source C projects.

Change patterns. By mining inserts and deletes of method invocations in software repositories Livshits and Zimmermann (2005) extracted likely usage and error patterns of method calls. Kagdi *et al.* (2007) used frequent sequential pattern mining including the syntactical context of where the call occurs to find call-usage patterns. An approach with a similar goal was provided by Wasylkowski *et al.* (2007) that does not require a software history for the analysis. Furthermore, it is not limited to sets of method calls but also supports any object usage patterns. All three approaches are similar in spirit to our method ranking. A combination of these approaches with our change information can provide corrections to usage patterns along with appropriate condition checks.

Also by mining inserts and deletes of method invocations Breu and Zimmermann (2006) extracted method call change patterns and identified cross-cutting changes. In contrast, we focus on the type of change solely without giving them domain specific semantics. We can complement their findings by distinguishing the introduction of aspects with further change types.

BugMem a tool developed by Kim *et al.* (2006b) mines bug fixes from software repositories to reconstruct pairs of bug and fix patterns. Using such project specific bug patterns, the approach locates error prone software parts. Compared to existing bug finding tools *BugMem* is not limited to a predefined set of bug patterns. We use a similar technique to extract relevant changes and bug fixes for our method ranking approach. The ranking together with the provided change information can be used to locate bugs, but this is not meant to be the primary purpose.

Automatic bug finding tools have a high false positive hit rate. Because of that Kim and Ernst (2007) used the change history of a software system to filter warnings provided by such tools.

Kim *et al.* (2007a,b) inference approach represents the changes concisely as first-order relational logic rules. Each of them combines a set of similar low-level transformations and describes exceptions that capture anomalies to a general change pattern. The change type patterns on the API level are similar to the patterns found by Kim *et al.*

2.4 Recommending Changes

One of the research directions to contribute to *support* software evolution is recommending changes during development. This research direction is rather new in the area software evolution analysis. We discuss recent and promising approaches.

Holmes *et al.* (2006) implemented the example recommendation tool *Strathcona*. It uses the structure of source code under development to find relevant examples in a repository. The found examples then assist a developer on how to use or extend an API.

Based on how a framework adapts to its own changes Dagenais and Robillard

(2008) developed a recommendation system that suggest replacements for framework elements accessed by client programs. The change information we provide can complement these suggestions. For instance, we can additionally suggest necessary condition checks before calling a recommended method.

Stoerzer *et al.* (2006) presented an approach for change classification that helps programmers identify changes responsible for test failures. Ren and Ryder (2007) described an approach with a similar goal. They used a heuristic based on calling structures of a failed test to rank method changes that might have affected that test. The advantage of the latter approach over the former is that it does not relay on all affecting changes for all affecting tests. It only requires the affecting changes of the failed test. Both approaches use the *Chianti* Eclipse tool presented by (Ren *et al.*, 2004) to extract the changes responsible for test failures.

2.5 Clustering Software Artifacts

Clustering is a technique for identifying items within a data set that belong together. It has received much attention in the field of software engineering. Typically the two problems that are adressed with clustering are feature location in source code and remodularization of legacy software systems.

Recently, Maqbool and Bari (2007) gave a survey on the current state-of-the-art of hierarchical clustering in software engineering and discussed how to evaluate results obtained with clustering. Koschke and Eisenbarth (2000) presented a framework for experimental evaluation of clustering techniques.

Feature location. The idea of *feature location* is to group source code artifacts that contribute together a specific feature of a system, *e.g.*, finding all classes that implement the network functionality of a system. It fits in the broader context of software comprehension and architecture recovery.

Maletic and Marcus (2001) clustered source code entities based on their linguistic vocabulary. They assumed there is an informal semantic logic on how developers choose names for variables, functions, procedures, etc., and how they write comments. Consequently, source entities that contribute to the same functionality will use the same vocabulary.

Marcus and Maletic (2001) used this idea of the underlying informal semantic in source code to detect high level concept clones. Similar work was done by Kuhn *et al.* (2007). They used the same linguistic approach to cluster source code by its vocabulary. They further provided an automated labeling mechanism to denote the implemented feature behind a cluster of source code entities.

Van Deursen and Kuipers (1999) identified objects in a Cobol application using clustering to support the migration to the object-oriented paradigma. Phattarsukol

and Muenchaisri (2001) applied clustering to identify candidate objects in procedural source code.

Anquetil and Lethbridge (1998) extracted concepts from file names based on clustering. Robillard and Murphy (2007) used concern graphs to address inadequate separation of concerns. A concern graphs models and documents which parts of the source code relate to a certain concern.

Our change type clustering approach aims to identify concepts in changes rather than in the source code itself.

Remodularization. The area of software remodularization focuses on restructuring existing legacy systems. It is a long-term research topic fundamental to reengineering, and received significant contributions. Clustering is a promising but not the only technique to restructure existing software. For instance, graph-based approaches, *e.g.*, Müller and Klashinsky (1988), or neural network approaches, *e.g.*, Schwanke and Hanson (1994), were also used in this research area. As we cluster change types that are applied frequently together over time, we concentrate our discussion on remodularization by means of clustering.

When using clustering to restructure an existing legacy system, clusters represent the new structural entities. Early work was done by Hutchens and Basili (1985). They used clustering with data bindings between routines to gain a view of the system modularization.

Anquetil and Lethbridge (1999) evaluated if clustering based on file inclusion in the `gcc` leads to "good software modules" by three quality criteria: Expert criterion, design criterion, and the size of the clusters.

Mancoridis *et al.* (1998) treated automatic remodularization as an optimization problem. They used hill climbing and genetic algorithms to cluster software components. They minimized the inter connectivity and maximized the intra connectivity between components.

Xu *et al.* (2004) used clustering to restructure a system at functional level. They considered cohesion as major factor concerning the quality of the structure of a system.

We are interested in finding change patterns that emerged over time to describe and understand how a system changed. These patterns could be the result of refactoring and remodularization efforts.

2.6 Résumé

With change distilling we aim at extracting fine-grained source code changes based on tree differencing in ASTs. One might ask whether it is necessary to use complicated tree differencing algorithms to extract changes. Change extraction is also

possible with other techniques as described in this chapter. It is possible, but using tree differencing in ASTs has two major advantages:

1. *Detection of move changes*. Our change distilling algorithm is able to extract move changes. Except for refactoring detection, all approaches are unable to detect *move* changes. But these changes are essential because they reflect context changes for source code entities in the method body. We rely on the extraction of move changes in Chapters 7 and 8.

2. *Parsing is necessary*. To augment change information with source code data, a parser is necessary. For instance, Maletic and Collard (2004) or Kim *et al.* (2006b) use parser to enrich textual differences with source code information. When using ASTs we can directly leverage source code information.

Except for refactoring detection, none of the existing change extraction approaches provide a complete taxonomy of source code changes. However, we are convinced that a taxonomy is essential to facilitate the discussion and reasoning about source code changes in the evolution of a software system.

By using change distilling to analysis change types we aim at providing additional insight into software evolution. Our strength is the reasoning of method body and method declaration changes:

1. We provide useful insights into the co-evolution of comments and source code. These insights allow us to reveal how developers maintain comments and to assess the commenting process in a software system. This contributes to existing comment-related research that analyzed traceability between documentation and source code, how much information special comments contain, and whether false comments have any impact on bugs.

2. We use clustering to identify hidden concepts in method body and method declaration changes. Identifying such concepts allows us to reveal different development activities and coding guideline shifts during the evolution of a software system. This contributes to existing evolution patterns studies that analyzed usage patterns, error patterns, and refactoring rules.

3. We highlight patterns of context changes for invocations to particular methods. Highlighting such patterns allows us to suggest context changes when invocations are changed. This contributes to existing approaches that recommend usage patterns, artefacts for change tasks, and adaptive changes for framework evolution.

II

Change Distilling

3 Change Types

OUR taxonomy of source code changes builds the basis for reasoning about continuing changes of a software system. The taxonomy has to define source code changes precisely so that we can build an algorithm to extract them. But, the taxonomy must still define the changes meaningfully, that means, they should be understandable by developers. Furthermore, the taxonomy must describe a classification of changes according to their possible impact on other source code entities and whether they may be functionality-modifying or functionality-preserving.

We present the result of defining and classifying source code changes. Our taxonomy defines source code change types with tree edit operations on the abstract syntax tree. It classifies these change types according to a change significance level scheme.

The remainder of this chapter is structured as follows: In Section 3.1 we describe the concepts and the taxonomy of source code changes. Section 3.2 defines the change types and assigns a change significance level to each of them. An example application of the change types with the change significance levels is presented in Section 3.3. We conclude this chapter with a reflection on the findings with respect to the hypotheses and research goals of this dissertation in Section 3.4.

3.1 The Taxonomy of Source Code Changes

The taxonomy of source code changes presented in this chapter focuses on object-oriented programming languages (OOPLs)—in particular on Java. By adjusting

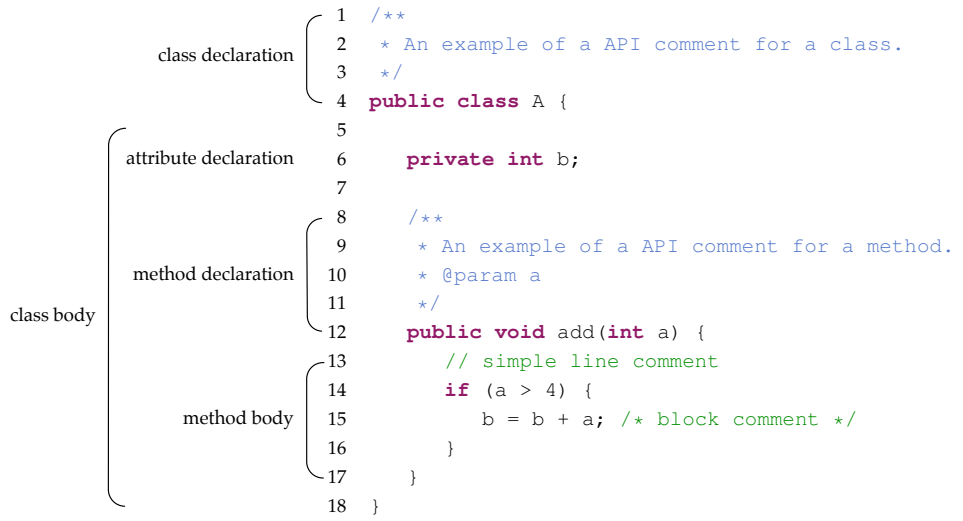


Figure 3.1: An example of a Java class

the change descriptions the taxonomy can also be used for other OOPLs.

In most OOPLs the concept *class* defines the framework for encapsulating functionality and state. Consider the example Java class in Figure 3.1. In our taxonomy, such a class is divided into *body*-parts: *class body*, Lines 5–18, and *method body*, Lines 13–17; as well as *declaration*-parts: *class declaration*, Lines 1–4, *attribute declaration*, Line 6, and *method declaration*, Lines 8–12. *API comments* (*e.g.*, Javadoc), Lines 1–3 and Lines 8–11 belong to the corresponding declarations. *Line comments*, Line 13, and *block comments*, Line 15, are part of the corresponding body. We are interested in changes in both parts, *e.g.*, the parameter type of the method declaration has changed or the assignment statement was moved outside an if-statement.

3.1.1 Level of Granularity

Our taxonomy of source code changes is based on the abstract syntax tree (AST). The smallest entities used are statements; structure statements such as loop or control structures are more coarse-grained than normal statements and are treated separately.

In an AST these source code entities are either sub-ASTs or leafs. For our taxonomy, ASTs consist of *entity nodes* with labels and values. The label represents the source code entity type, the value its textual representation depending on the entity type. For instance, the method invocation statement is a leaf node. The label

of the corresponding entity node is MI (Method Invocation) and the value is the method invocation as a string.

For the technical notation of entity nodes we use the terminology of Chapter 4. That means, for a node x, $l(x)$ denotes the label of x, $v(x)$ denotes the value of x, and $p(x)$ denotes the parent of x, if x is not the root. Children of an ordered entity tree node u are indexed, $\langle v_1, ..., v_m \rangle$, *i.e.*, a sequence of nodes. We call v_i the ith child of u.

We distinguish between ordered and unordered entity trees. In most object-oriented languages, such as Java or C++, methods or attributes of a class, *i.e.*, its children entity trees, do not have a particular order. Statements inside a method must have an order.

3.1.2 Basic Operations

Since ASTs are rooted trees and because source code changes transform an AST, the basis for source code changes are elementary tree edit operations. The detection of source code changes falls into the tree edit distance problem.

According to Valiente (2002) the *elementary tree edit operations* are *insert*, *delete*, and *substitute* of a tree node. Insert and delete operations are only allowed on leaf nodes. Substitution is a form of replacement of a tree node v with another, existing node w. In our taxonomy of source code changes, we use a weaker kind of substitution. The statements (*i.e.*, nodes) are not replaced with other statements, but their values are *updated*. Consider an if-statement with a certain condition. By changing the condition of the if-statement, the value of the entity node is *updated* with the new condition. The advantage of this terminology is that the entity tree (entity node and its subtrees) of the if-statement remains the same. The update operation is applied to the value of an entity node. That means, for instance, that changing the then-part of an if-statement is not an update of the if-statement.

A *move* operation can be described as a combination of an insert and a delete operation. In the context of source code changes we may give the move operation more importance, *e.g.*, changing the order of statements to gain performance or move a statement into an if-statement to check some conditions before executing it. We include the move operation in the set of elementary tree edit operations.

In our taxonomy we use the subscripts *old* and *new* to describe the values of the subscripted variable before and after the change. The taxonomy defines change types using the following operations. We provide an example for each operation in Section 4.1.2.

- **Insert**. $\text{INS}((l, v), y, k)$; insert new leaf node with label l and value v as kth child of node y.
- **Delete**. $\text{DEL}(x)$; delete node x from its parent $p(x)$.

- **Alignment**. MOV$(x, p(x), k)$; node x becomes the kth child of $p(x)$.
- **Move**. MOV$(x, y, k), p(x) \neq y$; node x becomes the kth child of y and is deleted from $p(x)$.
- **Update**. UPD(x, val); update $v(x)$ with val, *i.e.*, $val = v_{new}(x)$ and $v_{old}(x) \neq v_{new}(x)$.

Each change type is defined in the following format:
X *denotes the* Y Z.

"X" is written in small caps and is the name of the defined change type. "Y" is the name of the operation and "Z" the tree edit operation applying the change.

3.1.3 Change Significance Level Scheme

Our classification of source code changes defines how significant a certain change is. We define the *change significance level* of a change type as the possible impact of the change on other source code entities, *i.e.*, how likely is it that other source code entities have to be changed, when a certain change is applied. Additionally, whether a change may be *functionality-preserving* or *functionality-modifying* also influences the change significance level. If a change modifies the functionality of the enclosing entity, it is functionality-modifying, otherwise functionality-preserving. For instance, although a renaming strongly induces other changes, it does not (or should not) change the functionality of a program. Functionality-modifying or functionality-preserving relates to a single change. It is possible that a set of functionality-modifying changes is functionality-preserving for the corresponding program.

To classify the change significance level of a source code change, we use the levels *none*, *low*, *medium*, *high*, and *crucial*. Local changes, such as changes in a method body, are considered to have a low or medium change significance level, whereas changes on the interface of a class have a high or crucial significance level. For instance, if a parameter name of a method signature is changed, each access of this parameter inside the method body has to be changed. Indeed, such a change induces many other changes and according to its impact the change significance level is high. However, it does not change the functionality of the method. Therefore, we define the change significance level of parameter renaming as *medium*. Comment changes generally have the change significance level *none*.

3.2 The Change Types

In this section we present our set of source code change types. Each change type is defined as a single or a set of tree edit operations. For each definition, the

change significance level of a change is motivated and defined. We start with body-part changes from fine to coarse-grained entity types and finish with declaration changes. We use abbreviations for several source code entities: C for class; M for method, A for attribute. Other abbreviations are directly defined when we use them.

3.2.1 Body-Part Changes

Before describing the body-part change types in details we summarize them. The change significance level (change sign. lvl.) of several change types is split into *normal* and {*protected*, *public*} ({pro.,pub.}). Changes of source code entities defined as protected or public may have a stronger change impact on other entities than such defined as private. Change type names marked with * are functionality-preserving, all others are functionality-modifying.

		Change sign. lvl.	
Change type	Operation	normal	{pro., pub.}
Additional Functionality*	$\langle$ INS$((l(M), n(M)), C, k)$, INS$((l(P(M)), \{\}), M, 4)\rangle$	low	
Additional Object State*	$\langle$ INS$((l(A), n(A)), C, k)$ + INS$((l(T(A)), v(T(A))), A, 1)\rangle$	low	
Loop Condition	UPD$(l(L), v(CE_{new}(L)))$;	medium	
Expression Change			
Control Structure Condition	UPD$(l(CS), v(CE_{new}(CS)))$	medium	
Expression Change			
Else-Part Insert	INS$((l(EP), v(CS)), CS, 1)$	medium	
Else-Part Delete	DEL(EP)	medium	
Removed Functionality	DEL(M)	high	crucial
Removed Object State	DEL(A)	high	crucial
Statement Delete	DEL(s)	low	
Statement Insert	INS$((l(s), v(s)), y, k)$	low	
Statement Ordering Change	MOV$(s, p(s), k)$	low	
Statement Parent Change	MOV(s, y, k), $p_{old}(s) \neq p_{new}(s)$	medium	
Statement Update*	UPD(s, val)	low	
Comment Delete	DEL(CO)	none	
Comment Insert	INS$((l(CO), v(CO)), y, k)$	none	
Comment Move	MOV(CO, y, k), $p_{old}(CO) \neq p_{new}(CO)$	none	
Comment Update	UPD(CO, val)	none	

For body-part change we distinguish between method body changes and class body changes.

Method body changes

Method body changes are changes on the entity types control structure statement (CS), loop statement (L), and simple statement (s). Although programming languages may have different loop statements, they are all reducible to the simplest loop form (*e.g.*, while) and are not treated separately; similar for control structure statements, *e.g.*, switch-case statements can be expressed as if-statements.

For method-body changes we assume: (1) The parent of a statement s, $p(s)$, is either a statement or the method declaration, and (2) statements are ordered children of their parents, *i.e.*, if the statements $\langle s_1, ..., s_m \rangle$ are children of statement t, then s_i is the ith child of t.

STATEMENT ORDERING CHANGE: *A* statement ordering change *denotes the alignment operation* MOV$(s, p(s), k)$.

A statement ordering change may induce a change of the postcondition of the method and impact calling methods. However, changing the ordering of the statements may also be applied according to other functionality-preserving criteria, such as performance. We define the change significance level of the statement ordering change as *low*.

STATEMENT PARENT CHANGE: *A* statement parent change *denotes the move operation* MOV(s, y, k)*, with* $p_{old}(s) \neq p_{new}(s)$.

It is interesting to know, what the old and new parent of the statement is. Moving a statement from an if-statement to a loop may have more effect than from the else-part to the then-part of an if-statement. Since the parent of a statement and, accordingly, the label of the parent $l(p(s))$ is known, we are able to distinguish between different statement parent changes. As changing the parent of a statement has more impact on the postcondition of the method than a simple ordering change, we define the change significance level of a statement parent change as *medium*.

STATEMENT INSERT: *A* statement insert *denotes the insert operation* INS$((l(s), v(s)), y, k)$.

STATEMENT DELETE: *A* statement delete *denotes the delete operation* DEL(s).

STATEMENT UPDATE: *A* statement update *denotes the update operation* UPD(s, val).

Inserting and deleting statements are mostly functionality-modifying operations, whereas updating statements may also be applied because of renaming of a variable, parameter, or method. Nevertheless, we define the change significance levels of insert, delete, and update changes as *low* because they mostly impact their local environment only.

Instances of statement change types. With the label of the statement $l(s)$ and the one of its parent $l(p(s))$ we create instances of the statement change types, *i.e.*, statement source code changes. For instance, INS$((l(s), v(s)), y, k)$, with $l(s) = MI$, $v(s) =$ `foo.bar()`, and $y = (CS,$ `foo != null`$)$ is an instance of a statement parent change, *i.e.*, moving the method invocation statement `foo.bar()` into the if-statement with the condition `foo != null`.

Structure statements. In our tree representation of source code entities, we have chose the condition expression, $CE(\cdot)$, of a control structure or loop statements as the value of the corresponding entity node.

LOOP CONDITION EXPRESSION UPDATE: *A* loop condition expression update *denotes the update operation* UPD$(v(L), v(CE_{new}(L)))$.

CTRL. STRUCTURE CONDITION EXPRESSION UPDATE: *A* control structure condition expression update *denotes the update operation* UPD$(CS, v(CE_{new}(CS)))$.

Changing the condition expression is functionality-modifying and may also impact other changes inside the method body. We define its change significance level as *medium*.

A control structure has an alternative path, a so called else-part, EP, without a condition.[1] We use the term else-part also for switch cases of a switch-statement.

ELSE-PART INSERT: *An* else-part insert *denotes the insert operation* INS$((l(EP), CE(CS)), CS, k)$.

ELSE-PART DELETE: *An* else-part delete *denotes the delete operation* DEL(EP).

Inserting or deleting an else-part is functionality-modifying. We define the change significance level of those changes as *medium*.

Class body changes

We have already used and described changes on the body of a class (Fluri *et al.*, 2005). Inserting and deleting attributes and methods fall into this category.

[1] 'else if' is modeled as an else-part with a new control structure

ADDITIONAL OBJECT STATE: *An* additional object state *change denotes the insert operation* INS$((l(A), v(A)), C, k)$.

REMOVED OBJECT STATE: *A* removed object state *change denotes the delete operation* DEL(A).

An additional object state change has not any impact and is functionality-preserving. It has a *low* change significance level. A removing object state change is functionality-modifying and all accesses on the attribute have to be deleted, such a change has a *high* change significance level. Whenever the attribute has the access modifier *protected* or *public*, removing object state get the change significance level *crucial*.

ADDITIONAL FUNCTIONALITY: *An* additional object state *change denotes the insert operation* INS$((l(M), v(M)), C, k)$.

REMOVED FUNCTIONALITY: *A* removed object state *change denotes the delete operation* DEL(M).

Their change significance levels are defined analogously as for the change types of attributes: *low* and *high* or *crucial* in case the access modifier of the method is *protected* or *public*.

Since in most object-oriented programming language the entities in a class must not be ordered, move operations are not considered.

Comment changes

Line and block comments, CO, are treated equally for method and class body parts. The change types for comment changes do not distinguish between line and block comments.

COMMENT INSERT: *A* comment insert *denotes the insert operation* INS$((l(CO), v(CO)), y, k)$.

COMMENT DELETE: *A* comment delete *change denotes the delete operation* DEL(CO).

COMMENT MOVE: *A* comment parent change *denotes the move operation* MOV(CO, y, k), *with* $p_{old}(CO) \neq p_{new}(CO)$.

COMMENT UPDATE: *A* comment update *denotes the update operation* UPD(CO, val).

Comment changes do not impact any other change and are functionality-preserving. We define the change significance level for these changes as *none*.

3.2.2 Declaration-Part Changes

Before describing the declaration-part change types in details we summarize them. The change significance level (change sign. lvl.) of several change types is split into *normal* and {*protected*, *public*} ({pro.,pub.}). Changes of source code entities defined as protected or public may have a stronger change impact on other entities than such defined as private. Change type names marked with * are functionality-preserving, all others are functionality-modifying.

		Change sign. lvl.	
Change type	Operation	normal	{pro., pub.}
Adding Attribute Modifiability	DEL$(\mu_F(A))$	low	
Adding Class Derivability	DEL$(\mu_F(C))$	low	
Adding Method Overridability	DEL$(\mu_F(M))$	low	
Class Renaming*	UPD$(C, v(n_{new}(C)))$	high	
Decreasing Accessibility	INS$((l(\mu_A), v(\mu_A)), y, 1)$; DEL$(\mu_A)$; UPD$(\mu_A, val)$	high	
Attribute Type Change	UPD$(T(A), v(T_{new}(A)))$	high	crucial
Attribute Renaming*	UPD$(n(A), v(n_{new}(A)))$	medium	high
Increasing Accessibility	INS$((l(\mu_A), v(\mu_A)), y, 1)$; DEL$(\mu_A)$; UPD$(\mu_A, val)$	medium	
Method Renaming*	UPD$(M, v(n_{new}(M)))$	medium	high
Parameter Delete	DEL(ρ)	high	crucial
Parameter Insert	$\langle$ INS$((l(\rho), v(n(\rho))), P(M), k)$, INS$((l(T(\rho)), v(T(\rho))), \rho, 1)\rangle$	high	crucial
Parameter Ordering Change	MOV$(\rho, P(M), k)$	high	crucial
Parameter Type Change	UPD$(T(\rho), v(T_{new}(\rho)))$	crucial	
Parameter Renaming*	UPD$(\rho, v(n_{new}(\rho)))$	medium	
Parent Class Delete	DEL(T), $T \in p_{Cold}(C)$	high	crucial
Parent Class Insert	INS$((l(T), v(T)), p_C(C), k)$	crucial	
Parent Class Update	UPD$(T, v(T_{new}))$, $T \in p_{Cold}(C)$	crucial	
Parent Interface Delete	DEL(T), $T \in p_{Iold}(C)$	crucial	
Parent Interface Insert	INS$((l(T), v(T)), p_I(C), k)$	crucial	
Parent Interface Update	UPD$(T, v(T_{new}))$, $T \in p_{Iold}(C)$	crucial	

Change type	Operation	Change sign. lvl. normal	{pro., pub.}
Removing Attribute Modifiability	INS$((l(\mu_F), v(\mu_F)), A, 2)$	high	crucial
Removing Class Derivability	INS$((l(\mu_F), v(\mu_F)), C, 2)$	crucial	
Removing Method Overridability	INS$((l(\mu_F), v(\mu_F)), M, 2)$	crucial	
Return Type Delete	DEL$(T(M))$	high	crucial
Return Type Insert	INS$((l(T(M)), v(T(M))), M, 3)$, $T_{old}(M) = \{\}$	high	crucial
Return Type Update	UPD$(T(M), v(T_{new}(M)))$	high	crucial
API Comment Delete	DEL$(AC(x))$, $x = \{C, M, A\}$		
API Comment Insert	INS$((l(AC), v(AC)), y, k)$, $y = \{C, M, A\}$	none	
API Comment Update	UPD$(AC(x), val)$,$x = \{C, M, A\}$		

Each declaration part may have modifiers. Changes on modifiers can be declared in general. We distinguish between access and final modifiers. Static modifiers are currently not considered.

Access modifier changes

Access modifiers describe how restricted the access is to a class C, method M, or attribute A. We use the terminology of Java for access modifiers: $\mu_A = \{\text{private}, \kappa_A, \text{protected}, \text{public}\}$, with κ_A as the default access modifier, meaning that no access modifier is given.

Changes on access modifiers are defined once for all declaration-parts, because they have the same meaning for a class, method, and attribute.

INCREASING ACCESSIBILITY CHANGE: *An* increasing accessibility change *denotes the insert operation* INS$((l(\mu_A), v(\mu_A)), y, 1)$, *where either* $v(\mu_A) =$ protected *or* $v(\mu_A) =$ public, *and* $y = \{C, M, A\}$; *the delete operation* DEL$(\mu_A(x))$ *with* $v(\mu_A) =$ private, *and* $x = \{C, M, A\}$; *or the update operation* UPD$(\mu_A(x), val)$, *with* $x = \{C, M, A\}$, *where either* $v_{old}(\mu_A) =$ private *and* $val =$ protected $\vee$ $val =$ public, *or* $v_{old}(\mu_A) =$ protected *and* $val =$ public.

The increasing accessibility change is functionality-modifying. Its impact on other changes is rather low because accesses on an entity may remain unmodified. We define the change significance level of this change as *medium*.

DECREASING ACCESSIBILITY CHANGE: A decreasing accessibility change *denotes the insert operation* $\text{INS}((l(\mu_A), v(\mu_A)), y, 1)$ *with* $v(\mu_A) =$ private, *and* $y = \{C, M, A\}$; *the delete operation* $\text{DEL}(\mu_A(x))$, *where either* $v(\mu_A) =$ protected *or* $v(\mu_A) =$ public, *and* $x = \{C, M, A\}$; *or the update operation* $\text{UPD}(\mu_A(x), val)$, *with* $x = \{C, M, A\}$, *where either* $v_{old}(\mu_A) =$ public *and* $val =$ protected $\vee$ $val =$ private, *or* $v_{old}(\mu_A) =$ protected *and* $val =$ private.

Decreasing accessibility change has a deep impact on other changes. In the worst case all accesses on an entity have to be changed. The change significance level of this change is defined as *high*.

Final modifier changes

Besides access modifiers, a class, method, or attribute may also be declared as *final*, $\mu_F = \{\text{final}, \kappa_F\}$ where κ_F denotes the empty final modifier, meaning that no final modifier is given.

Possible final modifier changes are the basic operations insert and delete. Update or move are not reasonable for this modifier because it is a keyword that either exists or not.

FINAL MODIFIER INSERT: A final modifier insert *denotes the insert operation* $\text{INS}((l(\mu_F), v(\mu_F)), y, 2)$ *with* $y \in \{C, M, A\}$.

FINAL MODIFIER DELETE: A final modifier delete *denotes the delete operation* $\text{DEL}(\mu_F(x))$ *with* $x \in \{C, M, A\}$.

As the meaning of the final modifier is different for a class, a method, and an attribute, we give corresponding names for these changes:

- **Class**. The final modifier of a class declares the class as not derivable. We name the insert operation *removing class derivability* and the delete operation *adding class derivability*.
- **Method**. The final modifier of a method declares the method as not overridable. We name the insert operation *removing method overridability* and the delete operation *adding method overridability*.
- **Attribute**. The final modifier declares an attribute as unmodifiable. We name the insert operation *removing attribute modifiability* and the delete operation *adding attribute modifiability*.

Inserting the final modifier to one of the three entities is functionality-modifying and induces many changes, *e.g.*, deleting all write accesses to an attribute. We define the change significance level of this change as *crucial* for methods as well as

classes, and *high* for attributes. If the access modifier of an attribute is protected or public, the change significance level is also *crucial*.

Conversely, deleting the final modifier has no impact on other changes and is functionality-preserving. We define the change significance level for this change as *low*.

Attribute declaration changes

An attribute declaration A contains an access $\mu_A(A)$ and a final modifier $\mu_F(A)$, a type $T(A)$, and a name $n(A)$. The attribute initializer is not considered in this taxonomy.

ATTRIBUTE TYPE CHANGE: *An* attribute type change *denotes the update operation* $\text{UPD}(T(A), v(T_{new}(A)))$.

An attribute type change is functionality-modifying and implies changes on all accesses of the attribute—even worse when the attribute is public. We define the change significance level of this change as *high*, and *crucial* if the attribute is defined as protected or public.

ATTRIBUTE RENAMING: *An* attribute renaming *denotes an update operation of the name in an attribute declaration* $\text{UPD}(n(A), v(n_{new}(A)))$.

As attribute renaming also implies changes on all accesses of the attribute, but is functionality-preserving, its change significance level is *medium* and *high* if the access modifier is protected or public.

Method declaration changes

Besides the access and final modifier, a method declaration M contains an optional return type $T(M)$, a name $n(M)$, and a parameter sequence $P(M)$, $\langle \rho_1, .., \rho_m \rangle$. A parameter $\rho \in P(M)$ has a type $T(\rho)$ and a name $n(\rho)$. The combination of the name and the parameter sequence is called the method signature $\sigma(M)$.

RETURN TYPE INSERT: *A* return type insert *denotes the insert operation* $\text{INS}((l(T(M)), v(T(M))), M, 3)$ *with* $T_{old}(M) = \{\}$.

RETURN TYPE DELETE: *A* return type delete *denotes the delete operation* $\text{DEL}(T(M))$.

RETURN TYPE UPDATE: *A* return type update *denotes the update operation* $\text{UPD}(T(M), v(T_{new}(M)))$.

All three changes of the return type are, *i.e.*, functionality-modifying, and in most cases all method invocations of the changed method declaration have to be adjusted. We define the change significance level of this change as *high*, and *crucial* if the access modifier of the method is protected or public.

METHOD RENAMING: *A* method renaming *change denotes the update operation* $\text{UPD}(M, v(n_{new}(M)))$.

Method renaming is functionality-preserving, but all method invocations on the changed method declaration have to be adjusted. We define the change significance level of this change as *medium*, and *high* in cases where the method is defined as protected or public.

PARAMETER INSERT: *A* parameter change *denotes two insert operations* $\langle \text{INS}((l(\rho), v(n(\rho))), P(M), k), \text{INS}((l(T(\rho)), v(T(\rho))), \rho, 1)\rangle$.

PARAMETER DELETE: *A* parameter delete *denotes the delete operation* $\text{DEL}(\rho)$.

PARAMETER ORDERING CHANGE: *A* parameter ordering change *denotes the move operation* $\text{MOV}(\rho, P(M), k)$.

PARAMETER TYPE CHANGE: *A* parameter type change *denotes the update operation* $\text{UPD}(T(\rho), v(T_{new}(\rho)))$.

PARAMETER RENAMING: *A* parameter renaming *change denotes the update operation* $\text{UPD}(\rho, v(n_{new}(\rho)))$.

All parameter changes, except for parameter renaming, are functionality-modifying and induce changes on method invocations. We define the significance level of these changes as *high*, and *crucial* if the access modifier is protected or public. Parameter renaming is functionality-preserving and all accesses in the method body have to be adjusted. We define its change significance level as *medium*.

Class declaration changes

The class declaration C contains access $\mu_A(C)$ and final modifiers $\mu_F(C)$, a name $n(C)$, and parent classes $p_C(C)$, parent interfaces $p_I(C)$, or both.

CLASS RENAMING: *A* class renaming *change denotes the update operation* $\text{UPD}(C, v(n_{new}(C)))$.

As with other renaming changes discussed so far, the class renaming change is also functionality-preserving, but may induces a lot of changes. We define the change significance level of this change as *high*.

The inheritance concept in object-oriented programming languages is variably implemented. Some languages support multiple inheritance, such as C++ or Eiffel, other use interfaces instead, such as Java.

PARENT CLASS INSERT: *A* parent class insert *denotes the insert operation* $\text{INS}((l(T), v(T)), p_C(C), k)$.

PARENT CLASS DELETE: *A* parent class insert *denotes the delete operation* DEL(T) *with* $T \in p_{Cold}(C)$.

PARENT CLASS UPDATE: *A* parent class update *denotes the update operation* UPD($T, v(T_{new})$) *with* $T \in p_{Cold}(C)$.

The parent interface changes are defined accordingly. All of the defined parent class or interface changes are functionality-modifying and normally induce many other changes. We define their change significance level as *crucial*.

Because an interface declaration is a special kind of a class declaration, it is not treated separately.

API comment changes

API comments (*e.g.*, Javadoc), AC, are treated equally for method, attribute, and class declaration parts.

API COMMENT INSERT: *An* API comment insert *denotes the insert operation* INS($(l(AC), v(AC)), y, k$) *with* $y = \{C, M, A\}$.

API COMMENT DELETE: *An* API comment delete *change denotes the delete operation* DEL($AC(x)$) *with* $x = \{C, M, A\}$.

API COMMENT UPDATE: *An* API comment update *denotes the update operation* UPD($AC(x), val$) *with* $x = \{C, M, A\}$.

As an API comment strictly belongs to a particular declaration, move changes are not considered. API comment changes do not impact any other change and are functionality-preserving. We define the change significance level for these changes as *none*.

3.2.3 Limitations of the Taxonomy

When a renaming of a method parameter happens and the statements bound to this parameter are updated, the change on this statement must not be classified as proposed by Neamtiu *et al.* (2005). Such a statement change is functionality-preserving and does not have any impact on other changes. To improve the current situation, updated statements can be represented in more detail, *i.e.*, generating an entity tree of the statement, and calculate differences upon the entity trees. Additionally, slicing methods and program dependence graphs (Horwitz *et al.*, 1990) can be used to check the functionality-modifiability of a statement update.

In the current classification, changes on exception handlings are not yet considered as well. An interesting discussion on exception handling changes can be found in (Apiwattanapong *et al.*, 2007).

3.3 Application of Change Significance Level

We pursued a case study to demonstrate how the change significance analysis can help in understanding the evolution of source code (Fluri and Gall, 2006). The intention of the case study is rather to highlight the applicability of our taxonomy of source code changes than its complete validation—the validation of the change extraction is presented in Chapter 4. In particular we address the following question:

- To what extent are lines added and removed taken from the CVS log indicators for the significance of the applied changes?

We take the change history of two classes from the ArgoUML case study: `FigComment` with 65 revisions and `FigPackage` with 88 revisions.[2] We assigned the number 0 to 4 to the change significance levels *none* to *crucial*.

3.3.1 Lines Added and Removed as Change Significance

The change information provided by CVS are a textual diff (`cvs diff`) between two subsequent revisions f_{n-1} and f_n of a file f, and accordingly, the overall lines added and removed, *e.g.*, `lines: +15 -6`. For instance, this change information was used to compute code ownership in (Gîrba *et al.*, 2005). The number of lines added or removed may be an indicator for the significance of the changes between two subsequent revisions, meaning that the more lines were added, and removed respectively, the more the source code was changed. One of the drawbacks of this assumption is the high rating of text structure changes, such as indentation changes or the rearrangement of methods and attributes. In Figures 3.2 and 3.3 we show for each revision the lines added and removed as well as the sum of the change significance levels, total change significance for short.

FigPackage. The class (file) `FigPackage` has 88 (1.1–1.88) revisions. Examining the trend of the total change significance, three major changes stand out: Revisions 1.8, 1.49, and 1.81. Example major changes of lines added and removed that are not reflected in the total change significance are Revisions 1.16, 1.22, and 1.43. We discuss each of these revisions in more detail:

- In Revision 1.8 the class `FigPackage` doubled its size. A lot of statements and blocks of line comments were added. As we treat consecutive line comments as one block comment, inserting a number of line comments yields in one comment insert. Thus, the total change significance is lower than the number of lines added.

[2] Both classes are taken from the package `org.argouml.uml.diagram.static_structure.ui`

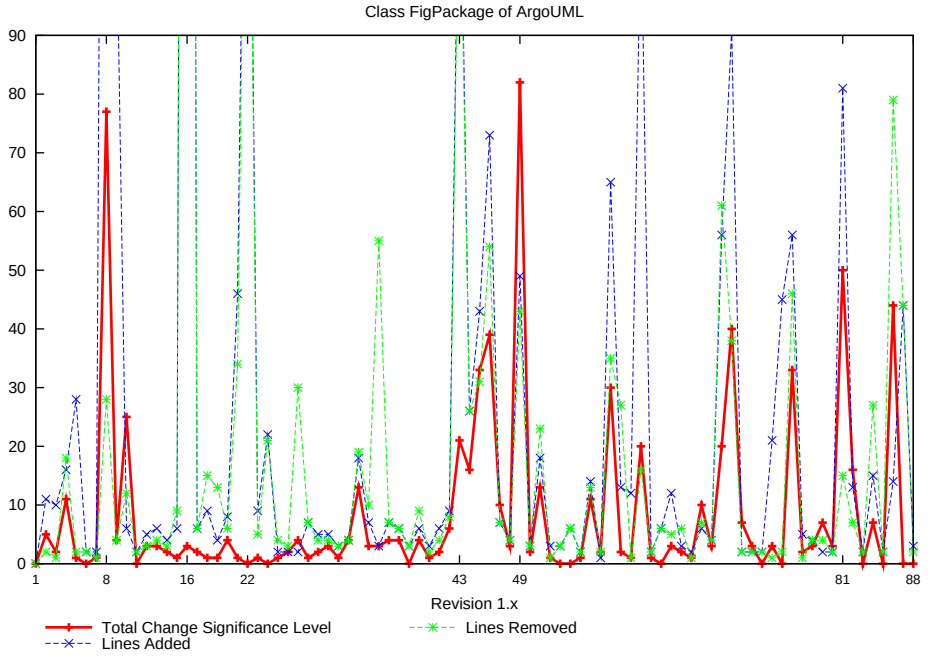


Figure 3.2: Difference of number of lines added/removed of CVS to change significance level of the `FigPackage` class from the ArgoUML software system

- In Revision 1.16 a branch was merged into the trunk. This accidentally caused a significant amount of indentation changes that are reflected in a high number of lines added and removed but in a relatively low total change significance. The indentation changes were undone in Revision 1.22.

- In Revision 1.43 an anonymous class was removed from a method and inserted as an inner class into the `FigPackage` file. That caused a high number of lines added and removed but only two change types.

- In Revision 1.49 a number of decreasing accessibility changes were applied to attributes. Each of these attributes was also renamed. That means, the change significance for such an attribute is 6, whereas the textual change is reflected in one line removed and one line added.

- In Revision 1.81 the discrepancy between the total change significance and the average number of lines added and removed is small. This is reflected in a number of simple method body changes.

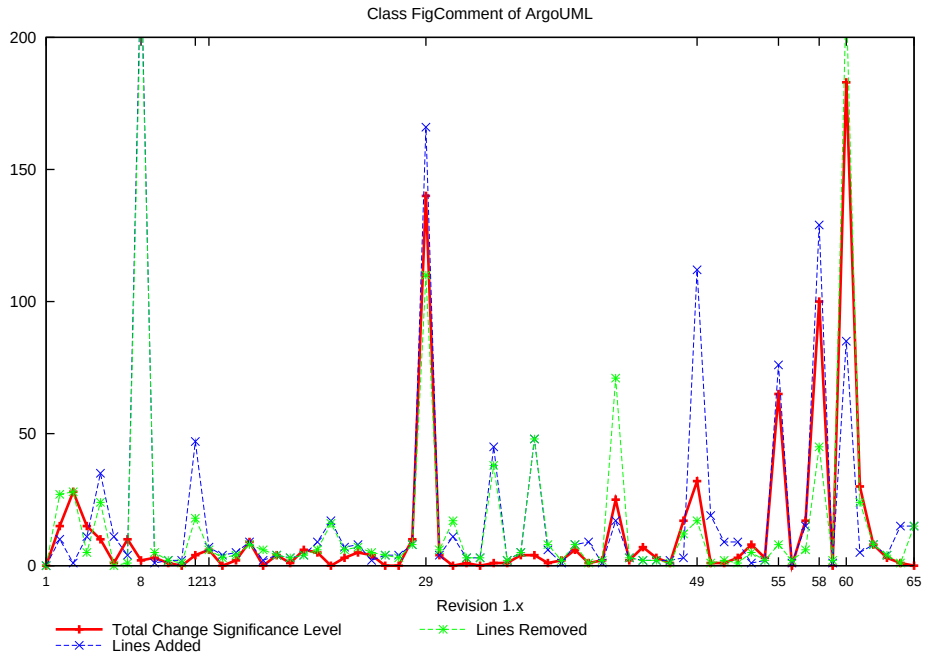


Figure 3.3: Difference of number of lines added/removed of CVS to change significance level of the `FigComment` class from the ArgoUML software system

For the change history of the class `FigPackage`, we conclude that the number of lines added and removed do not indicate the significance of the changes.

FigComment. The change history of `FigComment` has an interesting development towards the end of the observation period. The total change significance from revision 1.49 to 1.60 are increasing (with interrupts). The trend of the absolute sum of lines added and removed has a similar developing in this observation period—at least after 1.49. For this small period we observe that a relation between the total significance and lines added and removed recurs. On the other hand, we can also find the situation where no relation at all appears. Noteworthy are Revisions 1.8 and 1.49.

- In Revision 1.8 similar indentation changes happened as in Revision 1.16 of `FigPackage`.
- In Revision 1.49 a relatively long method was inserted into the class. A number of blocked line comments and some statements that cover multiple lines

were added to various methods. That leads to a smaller total change significance than the average number of lines added and removed.

For the change history of the class `FigComment`, we conclude that the number of lines added and removed do not indicate the significance of the changes.

Difference between the change significance and lines added/removed

In the examples of ArgoUML we have seen that the total change significance is either higher than lines added and removed or vice versa. We have already discussed the reasons for these discrepancies. Next we generalize the possible reasons.

Higher total change significance. Most of the change types are related to one or two textual line changes. For instance, a statement insert usually comprises one line addition. As the change significance level for statement insert is *low*, thus, 1, the number of lines added correspond to the change significance sum of all statement inserts.

However, a number of other change types exists that also comprise a single line change but have higher change significance levels. For instance, condition expression changes are classified as *medium*. Moreover, change types in declaration parts may have an even stronger impact on the difference between the change significance and lines added and removed. Assume the method declaration

```
public void calculate(int aNumber) {
```

changes to

```
public int calculate(long aNumber) {
```

The number of lines added and removed provided by diff, and CVS respectively, is for these changes 2. The effective changes are: Return type insert and parameter type change. Both change significance levels of these change types are *crucial*, thus, the total change significance is 8.

Lower change significance. The change significance is lower than lines added and removed if the textual changes have not been any source code changes. Examples for these cases are: (1) Indentation changes, (2) rearrangement of attributes, methods, or inner classes, and (3) licence term changes.

3.4 Résumé

We described our taxonomy of source code changes. It defines and names source code change types according to tree edit operations in the AST. We distinguished

between changes in the body and declaration parts of attributes, classes, and methods. The change significance level that is assigned to each change type reflects its possible impact on other source code entities and whether it may be functionality-preserving or functionality-modifying.

The taxonomy fullfils our requirement that it has to define change types precisely but should still be meaningful. For instance, the change type *parameter renaming* describes exactly its intention and is understandable for every developer. The use of a corresponding tree edit operations allows us to extract and classify tree edit operations in the AST automatically. The change significance level of the parameter renaming change type (medium) reflects that only source code entities in the body of the corresponding method have to be updated.

We have shown on a small example that the change significance levels give more meaning to the applied changes than already provided measures by CVS.

In this chapter we have shown that tree edit operations in the abstract syntax tree allow for a precise definition of source code change types. We have also shown that the type of a change indicates the impact that it may have on source code entities and whether it may be functionality-preserving or functionality-modifying. We therefore accept Hypothesis H1b as well as the first conceptual part (change definition) of Hypotheses H1a, and we regard the Research Goal G1 as fulfilled.

Extraction of Source Code Changes

4

SINCE source code can be represented as abstract syntax trees (AST), tree differencing can be used to extract detailed change information. This approach is promising because exact information of each source code entity is available in an AST. In this chapter we present our *change distilling* algorithm, a tree differencing algorithm for fine-grained source code change extraction and classification. We improved the existing algorithm for extracting changes in hierarchically structured data of Chawathe *et al.* (1996) to be applicable on ASTs. This algorithm finds changes according to the basic tree edit operations *insert*, *delete*, *move*, or *update* of tree nodes.

Our change distilling algorithm uses the *bigram string similarity* to match source code statements (such as method invocations, assignment statements, etc.) and the *subtree similarity* of Chawathe *et al.* to match source code structures (such as if-statements or loop statements). To further improve the matching, we use a best match algorithm for all leaf nodes and inner node similarity weighting. To overcome mismatch propagation in small subtrees we use dynamic thresholds for subtree similarity.

We developed a *benchmark* to evaluate source code change extraction algorithms. The benchmark consists of 1,064 manually classified changes in 219 revisions of eight methods from three different open source projects to evaluate our change distilling algorithm. Compared to the original change extraction algorithm of Chawathe *et al.*, we perform 45 percent better. We almost reach the minimum conforming edit script, *i.e.*, we reach a mean absolute percentage error of 34 percent.

The remainder of the chapter is organized as follows: In Section 4.1 we present

the original algorithm of Chawathe *et al.* and describe inadequacies concerning the extraction of source code changes. Section 4.2 presents string and tree similarity measures and our improved algorithm. In Section 4.3 the benchmark and our results are described. We conclude this chapter with a reflection on the findings with respect to the hypotheses and research goals of this dissertation in Section 4.4.

4.1 Change Extraction in Trees

Since source code is represented in a tree-like data structure, *i.e.*, in an abstract syntax tree (AST), we can use tree differencing algorithms to extract changes between two versions of a Java class. We use basic tree edit operations to describe changes applied to source code.

Our algorithm to extract changes is based on the work by Chawathe *et al.* (1996). In the following we introduce the terminology and outline their original algorithm, which outputs an edit script of basic tree edit operations transforming an original into a modified tree. Then, we illustrate why the original algorithm is not adequate for source code and discuss how we improved it to handle source code changes.

4.1.1 Terminology

Speaking in terms of graph theory, a tree is a directed acyclic graph consisting of nodes interconnected by edges representing a parent-child relationship in which each child has only one parent node. According to the notation used by Chawathe *et al.*, a node n is the *parent node* of a node m, $n = p(m)$, if m is a child of node n. Nodes along the path to the top of the tree are called *ancestors of* m. In return, m is called their *descendant*. The node in a tree that has no parent is called *root node* or *root*. Nodes that have no children are called *leaf nodes* or *leaves*. Nodes in between are *inner nodes*. Whenever the distinction between *root*, *inner node*, and *leaf* does not add to our discussion, we talk about *nodes* in general. A node n has a label, $l(n)$ and a value, $v(n)$. In our graphical tree representation, node labels are put inside a node, *e.g.*, *A*, and node values left or right beside the node, *e.g.*, "*val*." Figure 4.1 illustrates this terminology with an example tree.

Leaves in the tree are noncompound statements, *e.g.*, method invocation statements or assignment statements. For all nodes, the label is the type of the statement, *e.g.*, *MI* for a method invocation or *IF* for an if-statement. The value of an inner node depends on its label, for instance, the condition expression for if-statements: "*a* < *b*." For leaves, the value is the textual representation of the statement, *e.g.*, the method invocation statement *"x.foo(arg);"*.

Changes are detected between two trees T_1 and T_2. In general, T_1 denotes the original tree and T_2 the modified tree.

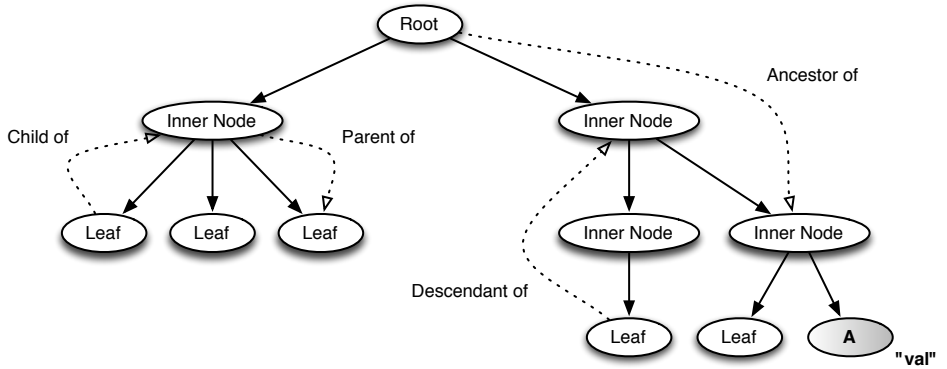


Figure 4.1: A generic tree structure. The rightmost leaf shows how we annotate labels and values of nodes

4.1.2 Basic Algorithm

Our change detection relies on the algorithm presented in (Chawathe *et al.*, 1996). Their algorithm detects changes in hierarchically structured data represented in tree-like data structures. To extract the changes, the algorithm splits the problem into two tasks:

- Finding a "good" matching between the nodes of the trees T_1 and T_2.
- Finding a minimum "conforming" edit script that transforms T_1 into T_2, given the computed matching.

Finding a "good," *i.e.*, correct and maximal, matching between the nodes is crucial to the outcome of the edit script task. The more nodes that can be matched, the better the minimum conforming edit script.

We first outline the calculation of the edit script and then describe the matching procedure in detail to highlight the parts to be adapted for detecting changes in source code.

Calculating an edit script

The matching set of node pairs is passed to the edit script generation that runs through five phases. Each phase is designed to detect one of the following basic tree edit operations, also illustrated in Figure 4.2.

- **Insert**. $\text{INS}((l, v), y, k)$; insert new leaf node with label l and value v as kth child of node y, *e.g.*, in Figure 4.2, H is inserted as child of B': $\text{INS}((H, \text{“”}), B', 2)$.

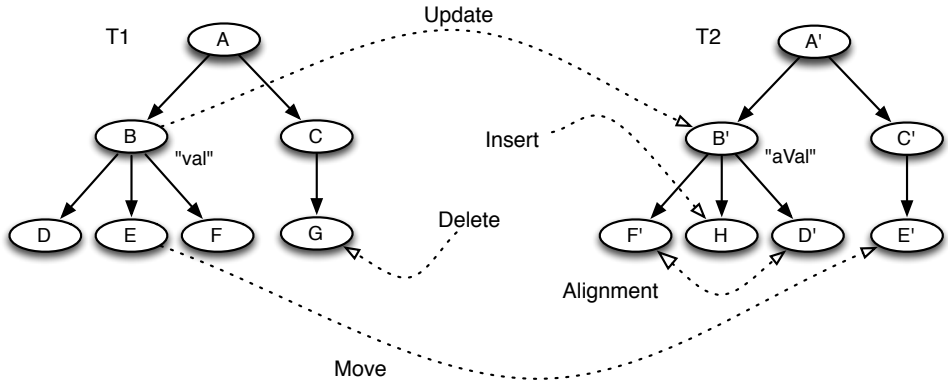


Figure 4.2: The five tree edit operations extracted by the edit script generation algorithm by Chawathe *et al.* Nodes with the same letter are intended to match (example: A matches A'). Node values have been omitted unless they changed from T_1 to T_2

- **Delete**. DEL(x); delete node x from its parent $p(x)$, *e.g.*, in Figure 4.2, G is deleted: DEL(G).
- **Alignment**. MOV$(x, p(x), k)$; node x becomes the kth child of $p(x)$, *e.g.*, in Figure 4.2, F' becomes the first child of its parent B' and D' becomes the third child of its parent B': MOV$(F, B', 1)$ and MOV$(D, B', 3)$.
- **Move**. MOV$(x, y, k), p(x) \neq y$; node x becomes the kth child of y and is deleted from $p(x)$, *e.g.*, in Figure 4.2, E is moved from B to C': MOV$(E, C', 1)$.
- **Update**. UPD(x, val); update $v(x)$ with val, *i.e.*, $val = v_{new}(x)$ and $v_{old}(x) \neq v_{new}(x)$, *e.g.*, in Figure 4.2, the value of B is updated: UPD$(B, \text{“}aVal\text{”})$.

Matching procedure

The matching procedure finds an appropriate matching set of pairs of nodes from T_1 and T_2. Chawathe *et al.* define two fundamental matching criteria necessary for the algorithm to produce a “good” matching set with which a minimum conforming edit script is achieved.

MATCHING CRITERION 1 (LEAVES)

$$match_1(x, y) = \begin{cases} true & \text{if } l(x) = l(y) \text{ and} \\ & sim(v(x), v(y)) \geq f \\ false & \text{otherwise} \end{cases}$$

Leaves match if their labels are equal and their values (as strings) are similar according to a given string similarity measure, $sim(x, y)$. The value f is the threshold for the string similarity. Pretesting the labels for equality is important to prevent the matching of different node types.

MATCHING CRITERION 2 (INNER NODES)

$$match_2(x, y) = \begin{cases} true & \text{if } l(x) = l(y) \text{ and} \\ & \frac{|common(x,y)|}{max(|x|,|y|)} \geq t \\ false & \text{otherwise} \end{cases}$$

where $|x|$ denotes the number of leaves contained by x. The inner node matching does not use similarities for the node values. Instead it uses a measure of how many leaves the subtrees have in common:

$$common(x, y) = \{(w, z) \in M | w \text{ is leaf of } x, \text{ and } z \text{ is leaf of } y\}, \text{where } M \text{ is the set of matched node pairs.}$$

The number of common leaves is put into proportion to the maximum number of leaves in either subtrees. The value t is the threshold for the inner node similarity. Matching Criterion 2 puts a strong focus on the leaves and is, therefore, good for LaTeX documents, where leaves (words or sentences of natural language) cover most of the text semantics.

Since the approach presented by Chawathe *et al.* is used for detecting changes in hierarchically structured documents, they use an assumption to make a *unique maximal* matching:

ASSUMPTION 1

For any leaf $x \in T_1$ there is at most one leaf $y \in T_2$ such that $sim(v(x), v(y)) > 0$.

The assumption that there is at most one leaf in T_2 that can match a corresponding leaf in the T_1 (and vice versa) is a necessary precondition for the algorithm to consequently produce an optimal matching and a minimal conforming edit script. Even if the assumption fails, Chawathe *et al.* apply a post-processing step to improve the solution. For source code comparisons, Assumption 1 is one of the main reasons why the approach by Chawathe *et al.* produces suboptimal results. In

Section 4.1.2, we discuss the assumption and the post-processing step, as well as the circumstances under which the post-processing step is insufficient for our concerns.

When matching fails

When applied to source code, the shortcomings of the basic algorithm impact the matching set—in these cases, the matching fails. However, *failing* does not mean that the algorithm yields incorrect results, *i.e.*, leading to an edit script that does not transform the original into the modified tree correctly. The edit script is always correct, but, if the matching is inadequate, the solution may not be minimal.

The quality of the sim-function and the associated threshold f, introduced in the first matching criterion, are crucial for an optimal matching on the leaf level. When Assumption 1 does not hold, a mismatch on leaves can be propagated to inner nodes, leading to a mismatch on higher levels. This can happen whenever a certain number of children of an inner node violate Assumption 1; this is particularly prominent for small subtrees. In the following, we discuss issues concerning leaf-matching based on node values and illustrate mismatch propagation.

Node values. Matching leaves is based on two conditions: First, the leaves have to be of the same kind, which we can verify by testing their labels for equality. The second condition applies to the values of the leaves and is evaluated using the function introduced in Matching Criterion 1. In terms of the AST that we use, values correspond to statements (or to the condition expression in case of an if-statement) that are strings. Consider the two strings *verticalDrawAction* and *drawVerticalAction*, which can be found, *e.g.*, in method invocation statements. From a human's point of view, we intuitively see that they can be considered as an original and a modified version of the same statement, especially when they were found in the same context, *i.e.*, in subsequent versions of the same method of a class.

Considering common string similarity measures, context semantics are missing. As we observed in our case studies, common renaming of identifiers during refactoring often involves changing the word order. To allow these strings to match, we have to lower the string similarity threshold, f, significantly, possibly resulting in false negatives in other places.

Small subtrees. A mismatch on a single leaf pair does not have a noteworthy impact on the quality of the outcome of the algorithm; we find additional insert and delete operations instead of update operations in the edit script. However, these mismatches can be propagated to higher levels of the tree, leading to a complete mismatch of a whole subtree and, therefore, to many unnecessary tree edit operations.

We discuss the propagation of mismatches using small trees as an example: Between the code snippets in Figure 4.3 (a) and (b), a single statement was deleted and a new one was inserted. The surrounding code did not change at all and the threshold t of Matching Criterion 2 is set to 0.6.

```
if (a > b) {
    foo.getHuga();
    foo.doNothing();
}
```
(a)

```
if (a > b) {
    foo.getHuga();
    foo.bar();
}
```
(b)

Figure 4.3: (a) The original if-statement; (b) The modified if-statement: The method invocation `foo.doNothing();` was replaced by `foo.bar();`

Figure 4.4 visualizes the same source code using an AST representation. The node with label *IF* denotes an if-statement. Its value corresponds to the if-condition. The node with label *THEN* denotes the then-part. The node with label *MI* denotes method invocation statements, that are listed as values.

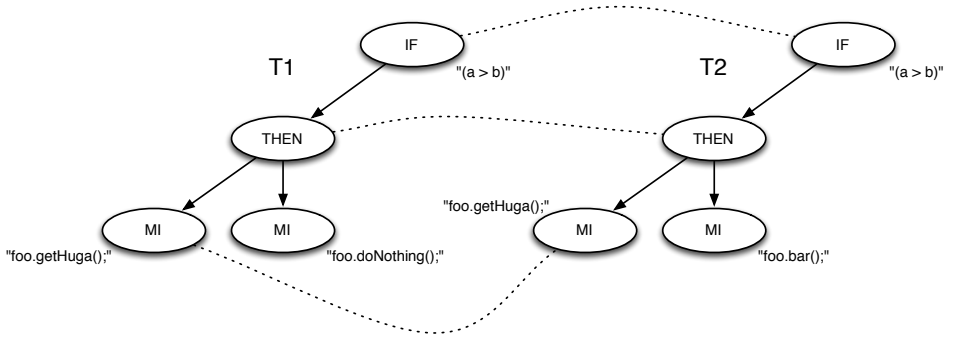


Figure 4.4: An example of two similar trees, T_1 and T_2, for which the algorithm fails to calculate a minimal edit script

For the matching, we traverse the trees bottom-up, *i.e.*, in depth-first manner from left to right. The leaves representing the method invocation `foo.getHuga();` in T_1 and T_2 match according to Matching Criterion 1. They are added to the matching set and marked as *matched*. Although the labels of both right leaves are the same, the values `foo.doNothing();` and `foo.bar();` cannot be matched. We proceed to the next level in the tree and reach the inner node representing the

then-part in T_1. Inner nodes are matched in accordance to Matching Criterion 2, we count the number of common leaf descendants of both nodes and divide them by the maximum number of leaves in either trees, leading to the tree similarity of 0.5 and, therefore, to a mismatch of the two then-parts:

$$\frac{|common(x,y)|}{max(|x|,|y|)} = \frac{1}{2} = 0.5$$

We proceed to the root of the subtree, the if-statement, which are not matched due to the inner node similarity of 0.5. The final (mis)matchings are shown in Figure 4.5.

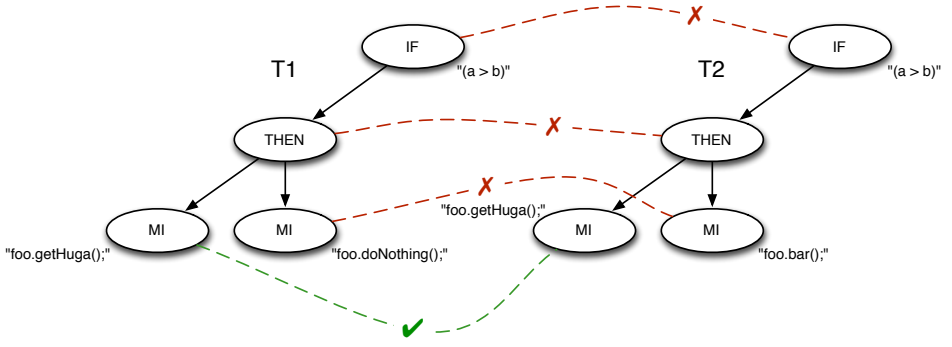


Figure 4.5: The whole subtree is considered as mismatched.

Although the trees in Figure 4.4 show a potential matching set of three node pairs, the algorithm fails—only one node can be matched using the matching criteria and a threshold of 0.6.

When Assumption 1 does not hold

Considering source code, similar statements can occur frequently. For instance, statements that print out a particular string on the console are commonly used for debugging. In such cases, there is more than one matching partner for a single node $x \in T_1$ leading to a violation of Assumption 1.

Figure 4.6 shows the consequences that a single statement insert (Node 3) can have: There is more than one possible counterpart in the right tree for Node 1, namely, Nodes 2 and 3. Since the tree is traversed in bottom-up manner, Nodes

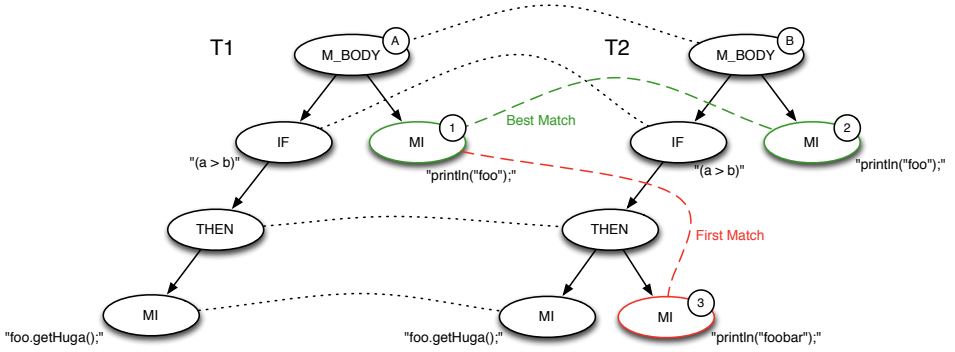


Figure 4.6: Suboptimal results are very likely to occur whenever Assumption 1 does not hold

1 and 3 are put into the matching set, whereas the better match, *i.e.*, the pair of identical Nodes 1 and 2, is not considered to match.

In T_1, the root is the only node that remains. Due to the simplicity of our example, we are able to catch mismatching propagation on this third level: According to Matching Criterion 2, the roots match because they have two common leaves divided by a maximum of three leaves in T_2, leading to a similarity of $\frac{2}{3}$, which lies above threshold $t = 0.6$. Even for our trivial example, the algorithm found nine changes—eight more than we have expected. We have expected the insert operation $\text{INS}((MI, \text{“}println(\text{”}foobar\text{”});\text{”}), THEN, 2)$, but the changes found are:

1. $\text{INS}((IF, \text{“}(a > b)\text{”}), M\_BODY, 1)$,
2. $\text{INS}((MI, \text{“}println(\text{”}foo\text{”});\text{”}), M\_BODY, 2)$,
3. $\text{INS}((THEN, \text{“”}), IF, 1)$,
4. $\text{INS}((MI, \text{“}foo.getHuga();\text{”}), THEN, 1)$,
5. $\text{MOV}((MI, \text{“}println(\text{“}foo\text{”});\text{”}), THEN, 2)$,
6. $\text{UPD}((MI, \text{“}println(\text{“}foo\text{”});\text{”}), \text{“}println(\text{“}foobar\text{”});\text{”})$,
7. $\text{DEL}((MI, \text{“}foo.getHuga();\text{”}))$,
8. $\text{DEL}((THEN, \text{“”}))$, and
9. $\text{DEL}((IF, \text{“}(a > b)\text{”}))$

In cases where Assumption 1 does not hold, a post-processing step is applied. For each matching pair (x, y), where $x \in T_1$ and $y \in T_2$, it is checked whether the matching partner of a child node c of x is a child node of y. If not, it is checked,

whether a child c' of y can be found such that $match(c, c')$ holds. In this case, the old matching pair is replaced by (c, c'). For further details, we refer to (Chawathe *et al.*, 1996). In the example above, the post-processing improves the matching set: For the matching pair $(\text{Node A}, \text{Node B})$, we check whether the matching partner of Node 1 is a child node of Node B. This is not the case. Therefore, we search for an unmatched child c' in Node B so that $match_1(\text{Node 1}, c')$ holds. Node 2 is such a c' in Node B. We replace the matching pair $(\text{Node 1}, \text{Node 3})$ with $(\text{Node 1}, \text{Node 2})$. The expected node is matched, which reduces the previous edit script by the changes {2, 5, 6}, but adds $\text{INS}((MI, \text{“}println(\text{“}foobar\text{”});\text{”}), THEN, 2)$.

There are a number of tree constellations in which the post-processing step does not improve the matching. In Figure 4.7, we show an example of such a constellation. Node 1 has been moved between T_1 and T_2 to a new position: It has been moved two levels up and is represented by Node 2 in T_2. Post-processing is not possible under these circumstances; the parent of Node 1 has no partner (corresponding) node in T_2.

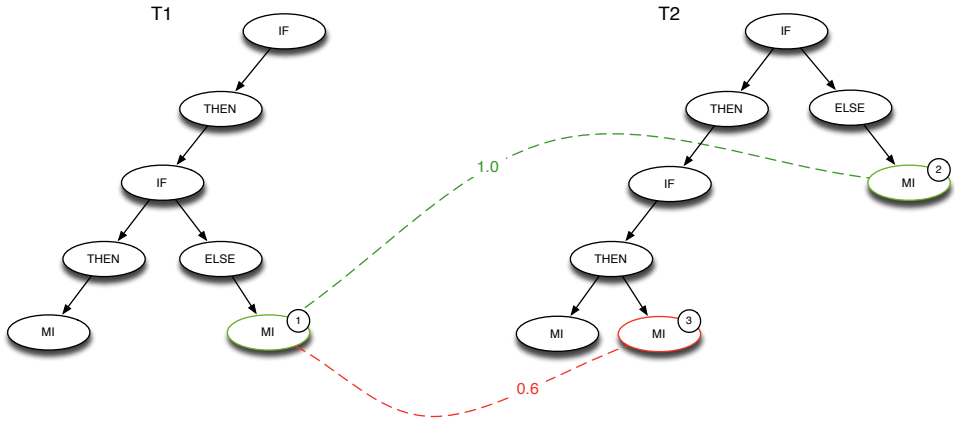


Figure 4.7: A trivial example of two trees, in which the post-processing step will not be able to improve matching

During our research on source code taken from open source projects such as ArgoUML,[1] we encountered mismatch propagations over two or three levels, *e.g.*, in nested if-then-else and try-catch statements. The levels of propagation seem to correlate with the nesting depth of, *e.g.*, if-statements or loop statements and the number of involved statements.

Despite their low frequency, these propagations can have huge implications on

[1] http://argouml.tigris.org

the size of the edit script and the classification of the occurred source code changes. In Section 4.2, we present how we overcome these inadequacies and customize the matching algorithm for detecting source code changes.

In summary, the shortcomings of the original algorithm for extracting source code changes are: (1) inadequate matching of node values, (2) using the *first* match instead of finding the *best* match, and (3) the propagation of mismatches in small subtrees. We have addressed these shortcomings and, next, we present a solution to improve the extraction of source code changes.

4.2 Change Distilling Algorithm

We stated that the hierarchical change extraction algorithm by Chawathe *et al.* needs to be adapted to take source code characteristics into account. In addition, we have discussed the circumstances under which the assumptions made for hierarchically structured text documents do not hold to compute a minimal edit script transforming an original AST into a modified AST (see Section 4.1.2). In this section, we discuss which parts of Chawathe *et al.*'s matching algorithm need to be customized for source code change extraction. Based on the desired improvements, we describe what measures and techniques overcome the inadequacy of the matching criteria discussed in the previous section.

To meet the requirements of source code change characteristics, we improve the original matching procedure with the following steps:

1. *Customize node value matching*. Since leaf matching is crucial to minimize the edit script, we aim at finding an adequate string similarity measure to match source code statements.

2. *Customize inner node matching*. We aim at finding a tree similarity measure that flexible matches inner nodes even if some unintended mismatches occur on the leaf level.

3. *Introduce best match*. Chawathe *et al.*'s Assumption 1 does not apply to source code because often multiple matching candidates for an original node are found. To address multiple matches, we select the leaf pair with the highest similarity.

4. *Use dynamic thresholds for inner node matching*. Propagation of mismatches leads to an enormous amount of unintended deletions and insertions. This is especially prominent for small subtrees—independent of the accuracy of the selected string similarity measure. Thus, we aim at finding a solution for matching small trees more adequately.

We proceed by developing similarity measures to reach the desired improvements. In the following, we discuss existing string and tree similarity measures that are adequate for source code and introduce our change distilling algorithm.

4.2.1 Matching of Leaves

Mismatches at the leaf level have tremendous impact on the size of the edit script. They can lead to mismatch propagation to higher levels in the tree and, consequently, to unnecessary node insert, delete, and move operations. String similarity measures that are robust to detecting common source code changes as well as techniques to reduce the amount of false first matches are crucial to overcome mismatch propagation.

We have evaluated string similarity measures provided by SIMPACK, a generic Java library for similarities and ontologies (Bernstein *et al.*, 2005). In this evaluation two measures were shown to be suitable for source code change extraction.

The Levenshtein string similarity measure

The Levenshtein Distance (Levenshtein, 1966) denotes the minimum number of operations needed to transform one string, s_a, into the other, s_b. These operations are (1) insert a character, (2) delete a character, or (3) substitute a character. The algorithm is based on the problem of the *longest common subsequence* (Hunt and McIlroy, 1976). A larger distance means that the strings are less similar, *i.e.*, that more operations are necessary to transform one string into another, whereas a distance of 0 operations denotes that the strings are equal. The runtime-complexity is $O(n \cdot m)$, where n is the number of characters in s_a and m in s_b.

For our concerns, distances are less useful than similarities since we cannot state that a distance of 3 is generally better than a distance of 4. It depends on the lengths of the compared strings. To overcome this situation, we normalize and convert the distance, using a distance-to-similarity conversion:

$$sim_{\mathrm{Lev}}(s_a, s_b) = 1.0 - \frac{D(s_a, s_b)}{D_{\mathrm{worstcase}}(s_a, s_b)}$$

The denominator $D_{\mathrm{worstcase}}$ is equal to the maximum costs experienced under the assumption that the longest common subsequence of s_a and s_b has a length of 0, *i.e.*, that they have no characters in common: $D_{\mathrm{worstcase}} = \max(m, n)$.

The Levenshtein Distance is susceptible to changes of word or character order. Consider the strings $s_1 = \mathit{verticalDrawAction}$ and $s_2 = \mathit{drawVerticalAction}$. If they are found at the same position in two versions of a source code entity, then it is

very likely that someone has performed a refactoring, *e.g.*, by unifying identifier nomenclature. The Levenshtein Distance does not recognize this similarity as our example illustrates: The longest common subsequence is "*verticalAction*." The remaining characters cause four insertions and four deletions, *i.e.*, a total of eight change operations and a distance of 8 respectively, leading to a string similarity of $sim_{\mathrm{Lev}}(s_1, s_2) = 1 - \frac{8}{18} \approx 0.56$.

Levenshtein Distance is inadequate in this case. Since we noticed during prototyping that a lot of unintentional mismatches on the leaf level were actually based on the deficiencies of the string similarity measure, we were eager to find an algorithm showing more robustness.

String similarity measures using n-grams

A family of string similarity measures is based on the *Dice Coefficient* (Dice, 1945)—a modification of the *Jaccard Coefficient* (Jaccard, 1912). Adamson and Boreham (1974) used the Dice Coefficient to rate the similarity of strings by setting their n-grams into relation. n-grams are bags and constructed by putting a sliding-window of length n over a string and extracting at each position the n underlying characters. For instance, the tri-grams of the string "vertical" are:

3-$grams$(vertical) = {"ver", "ert", "rti", "tic", "ica", "cal"}.

The n-gram similarity measure defined by Adamson and Boreham is the ratio of twice the number of shared n-grams and the total numbers of n-grams in two strings:

$$sim_{\mathrm{ng}}(s_a, s_b) = \frac{2 \times |n\text{-}grams(s_a) \cap n\text{-}grams(s_b)|}{|n\text{-}grams(s_a) \cup n\text{-}grams(s_b)|}$$

The Dice Coefficient with bi and trigrams is a popular word similarity measure. In combination with source code, bigrams have been used by Xing and Stoulia (2005a) for their *UMLDiff* approach, trigrams by Weidl and Gall (1998) for their *CORET* approach.

To illustrate the applicability of the n-gram similarity measure for source code change detection, we calculate the similarities for strings on which the Levensthein measure fails. As before, the strings to use are $s_1 = verticalDrawAction$ and $s_2 = drawVerticalAction$. The similarities for bi, tri, and four-grams are: $sim_{2\mathrm{g}}(s_1, s_2) = \frac{2\times14}{34} \approx 0.82$, $sim_{3\mathrm{g}}(s_1, s_2) = \frac{2\times12}{32} \approx 0.75$, and $sim_{4\mathrm{g}}(s_1, s_2) = \frac{2\times10}{30} \approx 0.67$. Using a hash-table to store the n-grams of both strings, the runtime complexity of the n-gram similarity measure is in $O(n + m)$—one order of magnitude faster than Levenshtein.

The n-gram similarity measure is more robust to changes to the word order, since it does not rely on the longest common subsequence. It primarily focuses

on common characters and secondarily on word order. Regarding source code in general and source code identifiers in particular, the measure allows for a more intuitive similarity scoring. During our experiments, the measure performed worse than Levenshtein only under rare circumstances (rare in conjunction with source code): It seems to be more susceptible to substitutions including misspellings due to phonetical reasons that are common in natural language but not so in source code. The strings *Levenshtein* and *Levnshtain*, for example, score with a similarity $\sim$0.72 when Levenshtein is used, but only with 0.5 when bigrams are used. Furthermore, the measure is limited to strings of a certain maximum length since the given number of different characters is finite. As a string gets longer, it will become more likely that most permutations between characters are covered. The number of character pairs in the intersection will therefore increase, leading to an imprecise similarity. However, we were not yet able to prove this expectation experimentally, but instead, we were able to confirm the effectiveness of the n-gram similarity measure to source code on the statement-level in our validation (see Section 4.3).

4.2.2 Similarity Rating for Best Match

As we have discussed in Section 4.1.2, Assumption 1 does not hold for source code represented in an AST. The post-processing step proposed by Chawathe *et al.* does not succeed either. Consequently, a *first* match cannot become a *best* match using the Assumption 1 and the post-processing step.

In general, a first match that is not the best match is formalized as follows: Let x be a leaf in T_1 and y be its matching partner in T_2. Furthermore, let z be another leaf in T_2 and f be the threshold, so that

$$sim(v(x), v(y)) \geq f \text{ and } sim(v(x), v(z)) \geq f \text{ but}$$
$$sim(v(x), v(y)) > sim(v(x), v(z))$$

Whenever z will be visited before y during postorder traversal, a suboptimal matching will be calculated.

Accordingly, we can derive a solution for that: Let x be a leaf in T_1. Furthermore, let mp_i be its ith possible matching partner in T_2, such that $i \in N$ and

$$sim(v(x), v(mp_i)) \geq f$$

We mark (x, mp_i) as *best match* until we find another possible partner $mp_{i+\epsilon}$, such

that $\epsilon \in N$ and

$$sim(v(x), v(mp_{i+\epsilon})) > sim(v(x), v(mp_i))$$

In this case, we mark $(x, p_{i+\epsilon})$ as *best match*. We repeat until we have tried to match all possible partners in T_2 to x.

The solution involves finding the matching partner $y \in T_2$ that matches $x \in T_1$ best. There are combinations of statements, so that x in T_1 has more than one possible partner, *e.g.*, when one and the same statement can be found over and over again in a block of code (for example, printouts for debugging). In this case, we apply the heuristics that unchanged statements stay in situ between subsequent versions of a source code entity: The first "best" match, *i.e.*, the matching pair with the highest similarity score that has been visited during postorder traversal first, will make it into the final matching set.

So far, we have developed an approach for finding the best partner $y \in T_2$ for leaf $x \in T_1$. However, this relationship is not always a two-way optimum, *i.e.*, x is not always the best partner for y. We can overcome this by calculating the similarity of each leaf pair $(x_i, y_j) \in T_1 \times T_2$ and add those pairs to the final matching set that show highest similarity.

4.2.3 Matching of Inner Nodes

Leaf matching propagates to inner nodes as similarity on inner nodes is calculated by the number of matching leaves. A measure for inner nodes that takes leaf matching into account and is robust to potential mismatches or small subtrees is important for a maximal matching set. Chawathe *et al.* presented a simple but adequate tree similarity measure for inner nodes (Matching Criterion 2). In this section, we discuss the suitability of this measure and other measures in terms of source code characteristics and small subtrees.

Tree similarity used by Chawathe *et al.*

The tree similarity measure used by Chawathe *et al.* (Matching Criterion 2) takes only descending leaves into account when deciding whether two nodes should match. Inner node descendants are ignored completely. This is an adequate approach for similarity analysis of structured text documents such as those that are written in LaTeX, where the inner nodes are used for structuring means and do not hold any semantics. For source code, inner nodes are more important since some of them cover fundamental constructs, such as iterations as well as alternatives or exception handling.

Since, for instance, an else-part may contain an if-statement, matching between descendants can occur. During our studies, it happened that an else-part matched with a descendant else-part. This matching resulted in a non-applicable move operation since a parent node cannot become a child node of one of its descendants. To overcome such situations, we added the check that the string similarity of the value of inner nodes must also satisfy the threshold t. Whenever a node does not have its own value, it inherits that of its parent to emphasize their affiliation.

Dice Coefficient for inner nodes

By using the Dice Coefficient, we get a measure taking inner nodes into account. In conjunction with code clone detection, Baxter *et al.* (1998) used the Dice Coefficient to calculate the similarity of two ASTs. For our purpose, we apply the same measure to inner nodes:

$$sim_{\text{Dice}}(T_a, T_b) = \frac{2 \times |nodes(T_a) \cap nodes(T_b)|}{|nodes(T_a) \cup nodes(T_b)|}$$

where $nodes(T_x)$ denotes all nodes of T_x including the root.

Taking inner nodes of the subtrees into account does not impact the value of the similarity measure, because the matching of leaves propagates to inner nodes. A more important aspect of the Dice Coefficient is that common nodes of T_a and T_b are weighted more than mismatches. When two trees share most of their nodes, but T_b differs in structure from T_a by a few changes, the Dice Coefficient is more robust than the measure used by Chawathe *et al.* Overall, our evaluation showed that the algorithm of Chawathe *et al.* including inner node similarity weighting and dynamic threshold (see next sections) performs better than the Dice Coefficient.

Inner node similarity weighting

According to our adapted Matching Criterion 2, the similarity of inner node values and the similarity of their subtrees influence the similarity for inner nodes likewise. Therefore, two inner nodes do not match either because their node values mismatch or they have too few leaves in common. Regarding if-statements or loop statements, a value, *i.e.*, condition expression, mismatch may cause a tremendous amount of unnecessary changes. We overcome this situation by weighting the common leaves function more than the similarity of values between inner nodes.

Inhibiting propagation of mismatches in small subtrees

The similarity measures for strings and for trees introduced in the previous sections reduce mismatching of single nodes but do not reduce them for small subtrees. Consider the code snippets in Figure 4.8. According to Matching Criterion 2, the similarity between the two then-parts of the if-statements is 0.5 (one shared node, two leaves), causing a mismatch of the then-parts and the if-statements.

To weaken the high impact that small changes can have on small subtrees, we dynamically lower thresholds for small subtrees; dynamically, meaning in regard to the size of the subtrees under investigation. We experienced adequate matching results for $t = 0.6$ if $n > 4$ and $t = 0.4$ if $n \leq 4$, where n is the number of leaf-descendants of the inner node.

Lowering thresholds for all inner nodes, no matter how many leaf descendants they count, injects undesired behavior into the algorithm: The amount of similar inner nodes increases by lowering the threshold leading to false matches.

```
if (cancelled()) {
   close();

}
```
(a)

```
if (cancelled()) {
   close();
   logger.debug("user has cancelled action");
}
```
(b)

Figure 4.8: (a) A small if-statement; (b) A logging statement has been added

4.2.4 Our Matching Algorithm Used for Change Extraction

In this section we present our improved tree differencing algorithm suitable to extract changes in source code. To recall, our improvements are:

1. Using bigrams as a robust string similarity measure that is able to cover common changes of source code identifiers.
2. Adding a similarity check of node values to Chawathe *et al.*'s tree similarity measure to solve the problem of descendant subtree matching.
3. Using inner node similarity weighting to reduce inadequate mismatches of condition expressions.
4. Introducing the best match algorithm to reduce the impact of Assumption 1.
5. Using dynamic thresholds to reduce the propagation of mismatches in small subtrees.

We evaluated combinations of the discussed string and tree similarity measures as well as best match, dynamic threshold, and inner node similarity weighting with our benchmark (see Section 4.3 for a detailed discussion). The following combination of measures and techniques performed best for extracting source code changes:

- For Matching Criterion 1 (Leaves) we use the bigram string similarity measure:

$$match_1(x,y) = \begin{cases} true & \text{if } l(x) = l(y) \wedge \\ & sim_{2g}(v(x), v(y)) \geq f \\ false & \text{otherwise} \end{cases}$$

  where $f = 0.6$.

- In addition to Matching Criterion 1, we take the *best* match for a leaf x instead of the *first* match.

- Matching Criterion 2 (Inner nodes) is extended by the check whether the values of the inner nodes are similar:

$$match_2(x,y) = \begin{cases} true & \text{if } l(x) = l(y) \ \wedge \\ & \frac{|common(x,y)|}{max(|x|,|y|)} \geq t \ \wedge \\ & sim_{2g}(v(x), v(y)) \geq f \\ false & \text{otherwise} \end{cases}$$

  where $f = 0.6$ and $t = 0.6$.

- We add inner node similarity weighting: If the string similarity of inner node values, *e.g.*, the condition of an if-statement, is less than the threshold f, but $\frac{|common(x,y)|}{max(|x|,|y|)} \geq 0.8$ holds, $match_2(x,y)$ is true.

- The threshold for the inner node similarity measure is adjusted dynamically for small subtrees: $n \leq 4 \rightarrow t = 0.4$.

The final algorithm is presented in Figure 4.9. The input to the algorithm are two labeled and valued trees T_1 and T_2. The algorithm first calculates a complete matching of all leaves (Lines 5–9). The leaf pairs are sorted (Line 10) according to their similarity and the best matches are added to the final matching set (Lines 11–15). At the end, the inner nodes are matched using dynamic thresholds (Lines 17–22). The output of the algorithm is a set of matching node pairs that is used by the edit script algorithm to compute the tree edit operations.

```
1: Input: trees T_1, T_2
2: Result: final matching set: M_final
3: M_final ← φ, M_tmp ← φ
4: Mark all nodes in T_1 and T_2 "unmatched"
5: for all leaf x ∈ T_1 and leaf y ∈ T_2 do
6:    if match_1(x, y) then
7:       M_tmp ← M_tmp ∪ (x, y, sim_2g(v(x), v(y)))
8:    end if
9: end for
10: Sort M_tmp into descending order, according to the leaf-pair-similarity
11: for all leaf-pair-similarity (x, y, sim_2g(v(x), v(y))) ∈ M_tmp do
12:    M_final ← M_final ∪ (x, y)
13:    Remove all leaf-pairs from M_tmp that contain x or y
14:    Mark x and y "matched"
15: end for
16: Proceed post-order on trees T_1 and T_2:
17: for all unmatched node x ∈ T_1, if there is an unmatched node y ∈ T_2 do
18:    if match_2(x, y) (incl. dynamic threshold and inner node similarity weighting) then
19:       M_final ← M_final ∪ (x, y)
20:       Mark x and y "matched"
21:    end if
22: end for
```

Figure 4.9: Our matching algorithm used for change extraction

The runtime analysis of the matching algorithm from Chawathe *et al.* has to be extended by the additional computation steps. Assume $n = \max(|T_1|, |T_2|)$, where $|T|$ is the number of leaves. The costs to compare two leaves is denoted by c. The matching of all leaves is in $O(n^2c)$, *i.e.*, $O(n^2)$, since we have to compare each possible leaf pair. Sorting the generated $O(n^2)$ matching pairs is in $O(n^2 \log n)$. For each pair that is added to M_{final}, the whole M_{tmp} has to be traversed at most once to remove all corresponding leaf pairs. Thus, building M_{final} for the leaves is proportional to $n^2(1 + c + \log n)$. The runtime complexity of inner node matching can be derived from the original work by Chawathe *et al.*: The number of inner nodes in T_1 and T_2 is denoted by m. Matching Criterion 2 can be computed for all inner nodes in $O(mn)$ (we refer to (Chawathe *et al.*, 1996) for more details). In addition, the value comparison of the inner nodes is in $O(mc)$. The overall runtime of inner node matching is $O(m(c+n))$. In summary, the total time of the matching

algorithm is proportional to

$$n^2(c + 1 + \log n) + m(c + n)$$

Compared to the original algorithm by Chawathe *et al.*, our runtime is $O(\log n)$ slower. We describe in Section 5.2.1 how we mitigate the impact of this additional factor to optimize the runtime performance of our change distilling algorithm.

4.3 Empirical Validation

In Section 4.2.4, we have described our change distilling algorithm. To investigate the quality of our improvements, we developed an extensive benchmark. The benchmark consists of a set of special test cases and of a large data set of manually classified changes. The data set is taken from three different open source case studies: ArgoUML,[2] Azureus,[3] and JDT.[4] With the benchmark, we show that our improvements approximate the minimum conforming edit script more closely than Chawathe *et al.*'s change detection algorithm. Although the CHANGEDISTILLER, our implementation of the change distilling algorithm (see Chapter 5), is able to detect changes at the class level as well, our benchmark focuses on changes at the method level. Since our major interest lies in the tree differencing part of our algorithm, changes at the method level are sufficient—they cover all tree structures that may occur in an AST.

4.3.1 Preliminaries

The final step of CHANGEDISTILLER is to analyze, consolidate, and classify the tree edit operations into *change types* (see Chapter 3). Change types are the most suitable data set for benchmarking our change distilling algorithm because they are an adequate measure for the quality of our algorithm and straightforward to implement and validate manually. Change types represent the kind of changes that a human will intuitively find when she compares two subsequent versions of a Java method. For example, she will recognize that a method invocation statement has been inserted into a method rather than thinking of the corresponding tree edit operation.

Taking two versions ($n - 1$, n) of a Java method, we count the occurrences of each particular change type manually. We, then, run the CHANGEDISTILLER on

[2] http://www.argouml.org
[3] http://azureus.sourceforge.net
[4] http://www.eclipse.org/jdt

the same pair of versions. For each version pair $(n-1, n)$ and each change type t, we calculate the mean absolute error ϵ_t and the mean absolute percentage error δ_t:

$$\epsilon_t = \frac{1}{k}\sum_{i=1}^{k}|x_i(t) - \tilde{x}_i(t)|, \quad \delta_t = \frac{1}{k}\sum_{i=1}^{k}\left|\frac{x_i(t) - \tilde{x}_i(t)}{x_i(t)}\right|$$

where $x_i(t)$ is the expected number of occurrences of change type t, $\tilde{x}_i(t)$ is the found number of occurrences of change type t, and k the number of version pairs in which t was expected or found. The smaller the difference between the number of change types classified manually and found by CHANGEDISTILLER, the smaller the error and the better we consider the performance of our algorithm.

For each version pair $(n-1, n)$, we calculate the mean absolute error ϵ and the mean absolute percentage error δ for the edit script:

$$\epsilon = \frac{1}{m}\sum_{i=1}^{m}|x_i - \tilde{x}_i|, \quad \delta = \frac{1}{m}\sum_{i=1}^{m}\left|\frac{x_i - \tilde{x}_i}{x_i}\right|$$

where x_i is the expected length of the edit script, $\tilde{x}_i$ is the found length of the edit script, and m the number of version pairs.

Before applying these measures to our change distilling algorithm, we have to discuss one shortcoming in terms of counting change types for the benchmark: We cannot evaluate exactly, where the change occurred, since we do not store its exact location in the benchmark, but rather in which version and method it was found. This means that we can tell that, *e.g.*, two statement inserts were found in method `foo()` between Version 1.11 and 1.12, but not whether the statements were, for example, inserted into a particular then-part or somewhere else. Performing a manual qualitative analysis on the whole data set instead of restricting ourselves to a quantitative validation, is barely feasible; we would have to determine the exact location in the AST for each change by hand to compare it to the output of our algorithm. For a sufficiently large set of changes, this is too time consuming and error prone.

To show that counting the occurrence of change types is sufficient nonetheless, we performed a qualitative validation on a randomly selected sample of the data

in our benchmark. For this, we have calculated precision and recall as follows:

$$Precision = \frac{\text{\# relevant changes found}}{\text{\# changes found}} \quad Recall = \frac{\text{\# relevant changes found}}{\text{\# changes expected}}$$

The selected sample contains 13 pairs of Java method versions comprising 120 expected changes. We compared each of the 151 changes found by CHANGEDISTILLER with the expected changes manually and obtained a precision of $\frac{118}{151} = 0.78$ and a recall of $\frac{118}{120} = 0.98$. Furthermore, we observed that the found edit scripts always transform the old into the new version of the Java methods correctly. Consequently, a recall < 1.0 denotes that our algorithm found changes that replace the ones that we expected. For instance, the method invocation statement

```
mParameter.setKind(MParameterDirectionKind.IN)
```
[5]

was updated with

```
ModelFacade.setKindToIn(mParameter)
```

but CHANGEDISTILLER found a corresponding *statement delete* and *statement insert* instead. A precision < 1.0 denotes that our algorithm found a non-minimal conforming edit script with *virtual* changes, *i.e.*, pairs of changes in the same edit script, of which the second reverts the first one and vice versa. Consider the example of source code in Figure 4.10, taken from our benchmark. For this, we manually classified four statement inserts (one if-statement insert and three method invocations). For this particular case, our change distilling algorithm extracts five statement inserts, one statement delete, and two statement parent changes, leading to an absolute error ϵ of 4 and a percentage error δ of 50% of the length of the edit script. Since the topmost if-statements (Line 1) share only two out of five leaves (Line 2 and 3 in (a) with Line 2 and 8 in (b)), Matching Criterion 2 is not satisfied, *i.e.*, they do not match. Therefore, the edit script contains the insert and delete operations of the topmost if-statement and move operations of the first and the last statement from the deleted to the reinserted if-statement. Applying these four changes does not transform the source code but, leads to a non-minimal conforming edit script.

Regarding the high recall, we claim that our algorithm at least finds the changes we expect. However, in certain cases, it finds a conforming edit script that is not minimal. If it finds fewer than expected changes, such as, statement updates, a set of corresponding changes are found instead (*e.g.*, in case of statement update: Statement insert and delete).

With our benchmark, we show that the output of our change distilling algorithm approximates the minimum conforming edit script more closely than the

[5] In method `addOperation(...)` in the class `org.argouml.uml.reveng.java.Modeller` between Revision 1.45 and 1.46.

```
1  if (matches.length == 0) {
2     fElements =
3        growAndAddToArray(
4           fElements, type);
5     return;
6  }
```

(a)

```
1  if (matches.length == 0) {
2     fElements =
3        growAndAddToArray(
4           fElements, type);
5     if(SelectionEngine.DEBUG){
6        System.out.print(
7          "SELECTION - accept type("
8        );
9        System.out.print(
10          type.toString());
11       System.out.println(")");
12    }
13    return;
14 }
```

(b)

Figure 4.10: (a) The original if-statement; (b) The modified if-statement of method `acceptSourceMethod(..)` of class `jdt.internal.core.SelectionRequestor`

original algorithm. Therefore, we only benchmark with the error measures.

4.3.2 Our Benchmark for Change Extraction

For the benchmark, we use a combination of dedicated test cases and data from three different case studies. We discuss how we have chosen the data and what preparation steps they have undergone.

Test cases

The test cases serve as validation for our improvements. We focused on testing string similarity measures, matching of small subtrees, and special issues on ordering changes. Test cases that failed with the original algorithm had to pass with the customized algorithm. For that, we have hard coded exact tree edit operations and their classification between two source code version of one class. For an in-depth discussion of these test cases we refer to (Würsch, 2006).

Collecting changes from existing software

Special test cases are well suited to investigate specific or theoretical issues. They are insufficient for claiming whether an approach applies to real-world problems

or not. Therefore, we decided to integrate data from the open source projects ArgoUML, Azureus, and JDT of Eclipse. Choosing representative test data among ∼4,900 classes was a challenge. We fed the projects into CHANGEDISTILLER with the original change extraction configuration and applied the following criteria to find appropriate Java classes.

- *A lot of changes over time, few changes between revisions*. We preferred classes that have 100–200 revisions and contain methods that show 10–20 changes per revision.
- *Method size*. We have chosen methods with 50–500 lines of code.
- *Nesting*. Methods that have nested if and loop statements are most interesting in terms of the small-subtree-problem.
- *Diversity of changes*. We preferred classes with different change types since we want to benchmark our algorithm in a broad variety of source code structures.

According to the above criteria, we located eight candidate methods in total—each one in a different class—that we integrated into our benchmark. We performed a checkout of every revision in which the selected methods experienced changes. Preparation of the classes was done by deleting all fields and methods except the chosen ones. During manual inspection, we finally classified 1,064 changes in a total of 219 revisions (see Appendix B for the details of the selection). To reduce evaluator bias, two of our group members have classified the changes independently and consolidated their findings.

4.3.3 Results and Discussion

In Section 4.2 we claimed our algorithm is better suited for source code changes than the original algorithm by Chawathe *et al.* In this section we present and discuss selected comparisons between different configurations of our change distilling algorithm, *i.e.*, we show how the different configurations perform against each other. We benchmark different combinations of the following:

- The original *first* match algorithm for leaves or our *best* match algorithm.
- Either the tree similarity measure suggested by Chawathe *et al.* or the Dice Coefficient are used for inner node comparisons.
- We dynamically lower the threshold t for inner nodes to 0.4 whenever the left and the right tree roots have four or fewer descendants.
- We either turn on or off inner node similarity weighting.

- We use either Levenshtein or n-grams similarity measure to match node values.

For the string similarity measures, we use f as the threshold variable, and t as inner node similarity threshold.

Benchmarking

We have conducted four runs with different configurations:

(a) Chawathe *et al.*'s original algorithm, Levenshtein as string similarity measure, $f = 0.7$ and $t = 0.6$, dynamic thresholds as well as inner node similarity weighting disabled.

(b) Chawathe *et al.*'s original algorithm, bigrams as string similarity measure, $f = 0.6$ and $t = 0.6$, dynamic thresholds as well as node similarity weighting disabled.

(c) Our best match, bigrams as string similarity measure, $f = 0.6$ and $t = 0.6$, dynamic thresholds as well as inner node similarity weighting disabled.

(d) Our best match, bigrams as string similarity measure, $f = 0.6$ and $t = 0.6$, dynamic thresholds as well as inner node similarity weighting enabled.

The minimum conforming edit script comprises 1,064 changes and the smaller the mean absolute error ϵ and the mean absolute percentage error δ, the better the performance of the algorithm. Table 4.1 depicts the results from Runs (a) and (b), Table 4.2 of (c), and (d), in the respective columns.

Run (a) In the first run, we found fewer statement updates and condition expression changes than expected with a mean absolute error ϵ of 0.96 and 1.02 between each pair of versions. In other words, the algorithm has missed, on average, approximately one statement update and condition expression change per pair of versions. As indicated by the ϵ values of statement inserts and deletes, the missed statement update and condition expression change are replaced by a pair of statement inserts and deletes. The accuracy of finding statement updates depends on the accuracy of the string similarity measure. The fewer statement updates, the more statement insert and deletes are found. Besides the string similarity measure, the accuracy of finding condition expression changes relies on the matching of inner nodes. Two if-statements match if their conditions (*i.e.*, values) match and if the inner node similarity satisfies the threshold t. Thus, matching small trees has an impact on condition expression changes. A mismatch leads to deletes of if-statements and else-parts with additional insert and ordering, parent, or both

		(a)			(b)		
Change Type	x	$\tilde{x}$	ϵ	δ	$\tilde{x}$	ϵ	δ
Cond. Exp. Change	91	51	1.02	0.58	64	0.89	0.44
Else-Part Delete.	9	32	1.04	0.08	28	1.17	0.11
Else-Part Insert	15	40	0.86	0.06	36	0.88	0.06
Method Renaming	1	1	0	0	1	0	0
Param. Delete	9	12	0.43	0.07	12	0.43	0.07
Param. Insert	16	20	0.29	0.04	20	0.29	0.04
Param. Ord. Change	0	19	2.71	0	19	2.71	0
Para. Renaming	3	1	0.67	0.67	2	0.75	0.5
Para. Type Change	1	1	0	0	0	1	1
Return Type Change	1	1	0	0	1	0	0
Return Type Insert	1	1	0	0	1	0	0
Stmt. Delete	144	371	2.28	0.3	283	1.96	0.29
Stmt. Insert	391	640	2.15	0.4	552	1.89	0.42
Stmt. Ord. Change	14	105	2.26	0.12	82	2.23	0.19
Stmt. Parent Change	86	185	1.84	0.2	194	1.87	0.21
Stmt. Update	282	216	0.96	0.34	318	0.72	0.19
Total	1,064	1696	3.27	0.79	1,613	2.91	0.72

Table 4.1: Benchmark results of the Runs (a) and (b) including the run-time performance in seconds, ϵ and δ per change type and edit script for each configuration

changes. On the other hand, when their conditions do not match but their subtrees, a mismatch occurs as well. The original algorithm is not able to match nodes accurately, leading to a mean absolute percentage error δ of 0.79 with additional 3.27 changes per version pair as depicted in Column (a).

Run (b) While evaluating the results of the initial run, we found that the outcome mainly relies on the string similarity measure and on the chosen threshold. We therefore lowered the threshold to $f = 0.6$ and used bigrams as string similarity measure instead of Levenshtein. The Column (b) of Table 4.1 illustrates that the number of statement updates increased tremendously compared to the number of condition expression changes—it even exceeded the expected number of statement updates. The reason for this increase is the flexibility of the bigram similarity measure, leading to statement updates instead of inserts and deletes. Configuration (b) reduced the overall ϵ from 3.27 to 2.91. This decreased the δ by 7% down to 72%.

		(c)			(d)		
Change Type	x	$\tilde{x}$	ϵ	δ	$\tilde{x}$	ϵ	δ
Cond. Exp. Change	91	58	0.92	0.47	85	0.58	0.24
Else-Part Delete	9	25	1.06	0.12	14	0.78	0.22
Else-Part Insert	15	33	0.78	0.07	22	0.41	0
Method Renaming	1	1	0	0	1	0	0
Param. Delete	9	11	0.33	0.08	11	0.33	0.08
Param. Insert	16	19	0.23	0.04	19	0.23	0.04
Param. Ord. Change	0	19	2.71	0	17	3.4	0
Param. Renaming	3	1	0.67	0.67	1	0.67	0.67
Param. Type Change	1	1	0	0	1	0	0
Return Type Change	1	1	0	0	1	0	0
Return Type Insert	1	1	0	0	1	0	0
Stmt. Delete	144	264	1.84	0.32	225	1.77	0.34
Stmt. Insert	391	533	1.76	0.38	494	1.54	0.32
Stmt. Ord. Change	14	83	2.08	0.11	73	2.23	0.08
Stmt. Parent Change	86	162	1.56	0.11	118	1.14	0.21
Stmt. Update	282	259	0.41	0.14	260	0.41	0.14
Total	1,064	1,471	2.2	0.52	1,343	1.64	0.34

Table 4.2: Benchmark results of the Runs (c) and (d) including the run-time performance in seconds, ϵ and δ per change type and edit script for each configuration

Run (c) To further improve the result, in particular to reduce the number of statement updates, we used our best match algorithm with bigrams. Column (c) of Table 4.2 shows the corresponding results. Using the best match algorithm reduces the number of statement updates and increases the condition expression changes. The advantage of best match is that it is less likely that correct statement inserts, deletes, or both, are replaced by updates, because better matches are taken for the matching set. Using best match improved the output of the algorithm significantly. We achieved a δ of 52%; thus, we further reduced the ϵ by 0.71 to 2.2.

Run (d) The results of the last and most influencing improvement, *i.e.*, our matching algorithm, are shown in Column (d) of Table 4.2. In particular, the *inner node similarity weighting* and *dynamic threshold* increased the number of condition expression changes. The number of statement inserts, deletes, and ordering changes as well as the else-part inserts and deletes were reduced. The reason for the decrease of those changes was that more if-statements matched and, therefore, fewer

statements were moved to a new if-statement.

Using the dynamic thresholds, we are able to get rid of the mismatch propagation in small subtrees. This led to an improvement of the overall δ by 8%. Enabling the weighting of the inner node similarity derived a further decrease of the δ by 10%.

Concerning the runtime, we observed a decrease between the Runs (a) and (b) as well as an increase between (b) and (c) or (d). The Levenshtein similarity measure used in Run (a) is an order of magnitude slower than the bigram similarity measure used in Run (b). The best match algorithm used in Run (c) and (d) is slower than first match used in Run (a).

Our change distilling algorithm, in particular the configuration we used in Run (d), reduced the mean absolute percentage error δ by 45% from 79% to 34% compared to the original algorithm. The number of additional changes found was reduced by 2.08 from 3.27 to 1.64 per pair of versions.

Further benchmark runs

We performed further benchmarking using the Dice Coefficient and other n-grams. We do not discuss these results in detail as they were not as promising as our configuration used in Run (d), but summarize them briefly: Using tri or four-grams instead of bigrams resulted in an δ of 38% and 40%. Since tri and four-grams are less flexible than bigrams, fewer statement updates occurred. The Dice Coefficient for inner node matching combined with the various configurations resulted in a minimum ϵ of 43% which is lower than the one that was achieved with the inner node similarity of Chawathe *et al.*

4.3.4 Limitations

Coming back to the results in Tables 4.1 and 4.2, our algorithm is still limited in finding the appropriate number of move operations. In particular, the performances of parameter ordering changes and statement ordering changes are modest. After an in-depth inspection of the benchmark results, we found that the method `acceptSourceMethod(...)`[6] was responsible for these outliers. Removing this method from the benchmark yielded a δ of 30%; this is a further improvement of 4%. The number of parameter changes was decreased to one and all declaration changes were extracted correctly.

Concerning body changes, the main reason for the few additional errors was also due to this method because it mainly consists of small nested if-statements and loop statements. Although we used our dynamic threshold approach, these

[6] in `org.eclipse.jdt.internal.core.SelectionRequestor`

small blocks were not matched because (1) the node similarities of those blocks fall below 0.4, and (2) the depths of their subtrees are mostly bigger than 4.

Furthermore, the best match approach may match reoccurring statements that are not at the same position in the method body. For instance, consider that the first statement of a method changed, but the same statement reoccurs at the end of the method and stays unchanged. The best match approach will match the first with the last statement leading to a mismatch for the first statement. Such a mismatch can have, as in this particular case, tremendous impact on the extraction of other changes. We noticed that such mismatches led to replacements of nested if-statements and loop statements.

```
void acceptSourceMethod(
  IType type,
  char[] selector,
  char[][] parameterPackageNames,
  char[][] parameterTypeNames)
```
(Revision 1.35)

```
void acceptSourceMethod(
  IType type,
  char[] selector,
  char[][] parameterPackageNames,
  char[][] parameterTypeNames,
  boolean isDeclaration,
  int start,
  int end)
```
(Revision 1.39)

Figure 4.11: Parameter changes from Revision 1.35 to 1.39 of `acceptSourceMethod(...)`

The declaration changes, in particular, the parameter ordering changes are also an implication of the small tree problem. The parameter changes in Figure 4.11 happened from Revision 1.35 to 1.39 of the `acceptSourceMethod(...)` described above; three new parameters were inserted. The similarity between the parameter list nodes is 0.57 ($\frac{4}{7}$); thus the nodes do not match. This mismatch yields to the changes:

1. deletion of the old parameter list,
2. insertion of a new parameter list,
3. insertion of the three new parameters, and
4. moving of the existing parameters to the new list.

Besides the three parameter insertions, four further parameter ordering changes are classified—the parameter list insert and delete are omitted.

As we have selected the methods for the benchmark randomly and the δ of our algorithm is for all methods about 30%, except for the method described above,

we claim that unsolvable small tree problems occur relatively seldom. However, further investigations of this issue are needed and subject to future work.

4.3.5 Summary of Validation

To validate our improvements, we established an extensive benchmark comprised of 1,064 manually classified changes. Compared to the original algorithm of we approximate the minimum conforming edit script with a mean absolute error of 1.64 and a mean absolute percentage error of 34% per version pair, *i.e.*, an improvement of 45%. This means, on the average, we find less than two additional change types, whereas the original algorithm finds more than three additional change types between two versions. The results showed that the combination of our best match algorithm with bigrams, Chawathe *et al.*'s node similarity measure, dynamic thresholds, and the inner node similarity weighting achieved the best benchmark results.

Although our dynamic thresholds noticeably inhibit mismatch propagation in small subtrees, we consider the problem as not fully solved yet as the changes in method `acceptSource-Method(...)` showed.

4.4 Résumé

To overcome the imprecise results of textual differencing, we presented the change distilling algorithm for fine-grained source code change extraction. We enhanced the existing tree differencing algorithm of Chawathe *et al.* to classify source code changes according to our taxonomy of source code changes with the following substantial improvements:

- Using bigrams as a robust string similarity measure that is able to cover common changes on source code identifiers.
- Adding a similarity check of node values to Chawathe *et al.*'s tree similarity measure to solve the problem of descendant subtree matching.
- Using inner node similarity weighting to reduce inadequate mismatches of condition expressions.
- Introducing the best match algorithm to reduce the impact of Assumption 1.
- Using dynamic thresholds to reduce the propagation of mismatches in small subtrees.

Furthermore, we introduced an extensive benchmark to evaluate source code change extraction algorithms. The benchmark consists of 1,064 manually classified changes in 219 revisions of eight methods from three different open source

projects. By applying the benchmark to the CHANGEDISTILLER, the implementation of our change distilling algorithm (described in Chapter 5), we achieved significant improvements in extracting change types: Our algorithm approximates the minimum edit script by 45 precent better than the original change extraction approach by Chawathe *et al.* We were able to find all occurring changes and almost reach the minimum conforming edit script, *i.e.*, we reach a mean absolute percentage error of 34 percent, compared to 79 percent reached by the original algorithm.

Although our dynamic thresholds significantly inhibit mismatch propagation in small subtrees, we consider the problem as not fully solved yet. In our benchmark, we experienced such inadequacies with one particular method that is deeply nested and has major declaration changes. Since further improvements of string similarity measures are limited, we will investigate post-processing steps to filter further inadequate matches.

In this chapter we have shown that a tree differencing algorithm applied to pairs of abstract syntax trees allows for the extraction of source code changes. We therefore accept the second conceptual part (change extraction) of Hypothesis H1a, and we regard the conceptual part of Research Goal G2 as fulfilled.

5 ChangeDistiller

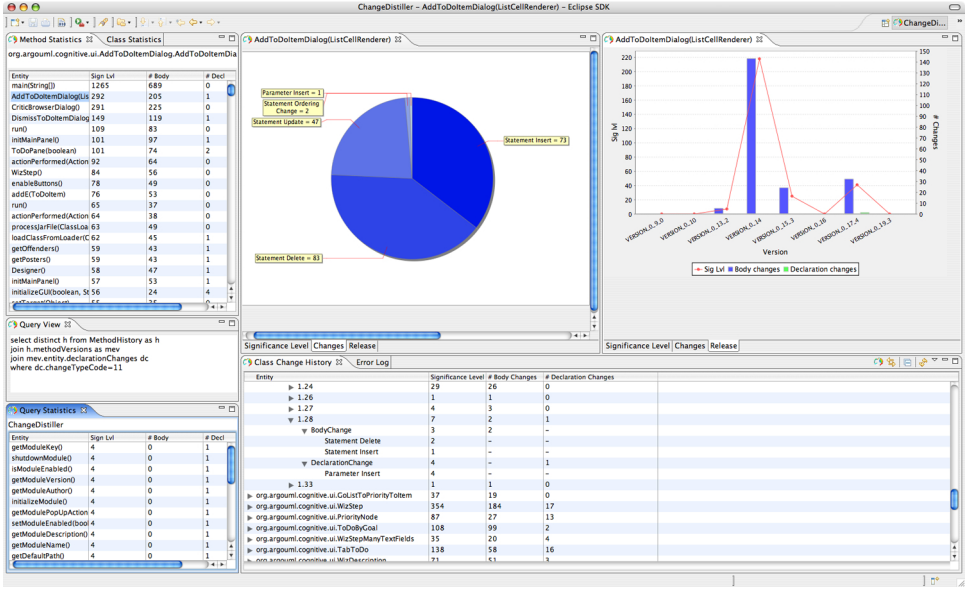


Figure 5.1: CHANGEDISTILLER in action

We have integrated the change distilling algorithm into the Eclipse integrated development environment (des Rivières and Wiegand, 2004). The integration allows us to extract change types and build a change history model of the evolution of a software project that is developed under Eclipse. CHANGEDISTILLER is the resulting tool of this integration, a tool that extends the Eclipse platform as a plugin (see the screenshot in Figure 5.1). The advantage of plugging CHANGEDISTILLER into Eclipse is threefold. First, Eclipse is a well-known IDE and used in open-source as well as in commercial environments. Contributing to Eclipse may have the effect that people start using our tool.

Second, being part of Eclipse as a plugin enables the access to all development functionalities of Eclipse. That means, for the change distilling algorithm we can leverage the AST generation.

Third, Eclipse is designed with and for the Java programming language. The tools Eclipse provide are tailored for Java. As the number of Java software systems increased over the last decade we obtain an exhaustive corpus of case studies.

The remainder of this chapter is structured as follows: Section 5.1 describes the architecture of the CHANGEDISTILLER. In Section 5.2 we present the change extraction process that CHANGEDISTILLER performs and the change history meta model of which CHANGEDISTILLER generates instances. We conclude this chapter with a reflection on the findings with respect to the hypotheses and research goals of this dissertation in Section 5.3.

5.1 Architecture of CHANGEDISTILLER

The architecture of CHANGEDISTILLER is depicted in Figure 5.2. It is built on top of the EVOLIZER platform to leverage historical data from CVS. It plugs our *change history meta model* into the EVOLIZER persistency layer to integrate versioning with change data.

CHANGEDISTILLER further relies on the Java Development Tools (JDT),[1] and compare functionality of Eclipse. JDT provides AST generation of Java classes and also visitors to navigate through an AST. The compare plugin of Eclipse provides textual and hierarchical differentiation of Java classes but only on a coarse-grained level.

In this section we describe the data we use from CVS and present shortly the EVOLIZER platform.

[1] http://www.eclipse.org/jdt

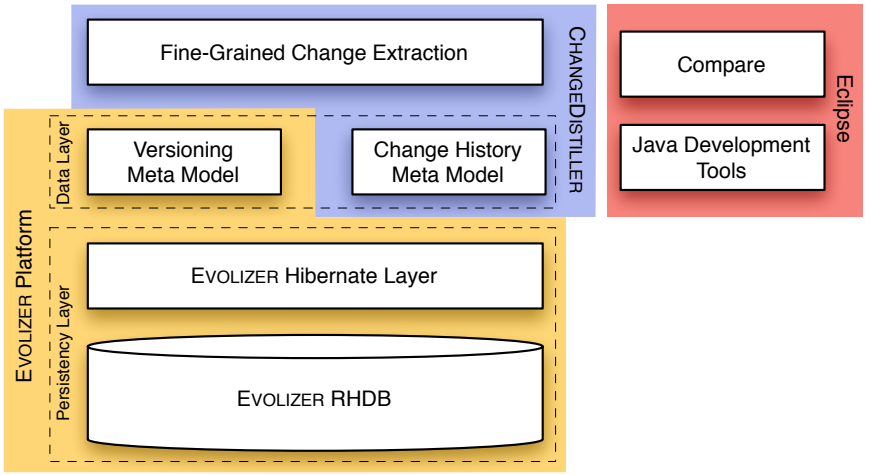


Figure 5.2: CHANGEDISTILLER as part of the EVOLIZER platform

5.1.1 CVS as Data Source

To extract the source code changes we rely on historical data from versioning systems. For this dissertation we use information obtained from CVS repositories. The data that CVS repositories provide are version-control files. In CVS terminology a file version is called a *revision*. CVS stores additional information for each revision in a repository: The date of the revision, the author that stored the revision, and an optional message that can be used to describe the revision. This additional information is called a *modification report*. When a new revision is stored into the CVS repository we speak about a *commit*. In CVS terminology a revision emerges when a changed file is committed to the CVS on a specific date, by an author (developer), and along with a commit message. CVS provides the *log* functionality to show the change history of each file including revision number, date, author, and commit message for each revision of the file. For change distilling, we are interested in the changes between subsequent revisions of a Java class.

5.1.2 EVOLIZER

We have developed EVOLIZER, a platform to enable software evolution analysis within Eclipse. It is comparable with *Kenyon* (Bevan *et al.*, 2005) or *eROSE* (Zimmermann *et al.*, 2005). In our EVOLIZER the CVS terminology is mapped into an object-

oriented meta model; the *versioning meta model* (see Appendix C). The EVOLIZER and the versioning meta model are not contributions of this dissertation, thus, we only describe their intent briefly. The EVOLIZER RHDB is the database to store the extracted historical data of a software system. The RHDB concept is presented by Fischer *et al.* (2003b). The Hibernate[2] layer of EVOLIZER maps object-oriented models to the RHDB. Besides the object-to-relational mapping, the Hibernate layer allows us to add new meta models to the EVOLIZER platform. By integrating our change history meta model (described in Section 5.2.4) into the Hibernate layer we can link change histories to versioning data.

To populate the RHDB, EVOLIZER parses the log of a CVS repository, builds up a versioning model, and stores it via Hibernate into the RHDB. During this process the content of each file revision is also stored into the RHDB for faster change extraction.

5.2 Fine-Grained Change Extraction Process

We have integrated the change distilling algorithm into Eclipse. Starting with an Eclipse project, CHANGEDISTILLER is able to extract changes from the version chain of a single class, packages, or a whole project. Figure 5.3 depicts the change extraction process of CHANGEDISTILLER. We start with an already populated EVOLIZER RHDB.

5.2.1 Preprocessing Changed Entities

From a project, revisions of Java classes are fetched from the EVOLIZER RHDB. For two subsequent revisions of a Java class we use the compare plugin to extract the methods and attributes that have changed (see Index 1 in Figure 5.3). This prefiltering step leads to smaller trees for comparison. Assume a class has about 1,000 lines of code, but only a single method with 20 lines of code has changed. Using the compare plugin reduces the input to our change distilling algorithm significantly. The complexity of the compare plugin is in $O(n^2)$ where n is the number of members of a Java class. Recalling the runtime complexity of the matching algorithm, this is a considerable performance gain as the input trees are kept small.

5.2.2 Intermediate Tree Generation

For both versions of a changed method or attribute intermediate ASTs are created using the AST visitor from JDT (see Index 2 in Figure 5.3). Creating intermediate trees is necessary since the matching algorithm expects labeled and valued nodes

[2]http://www.hibernate.org

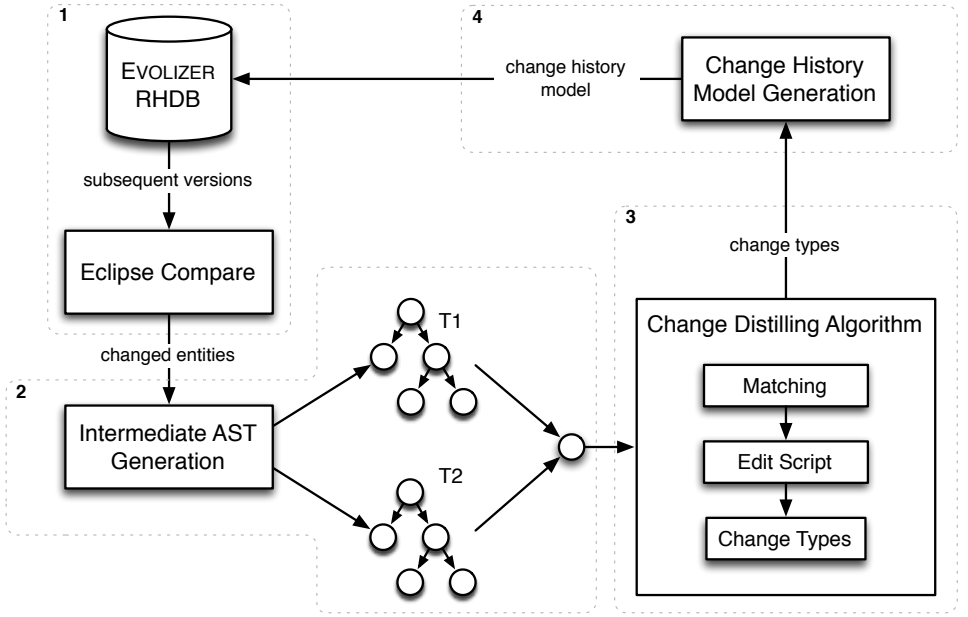


Figure 5.3: Fine-grained change extraction process

as well as a uniquely defined parent child relationship between hierarchically situated nodes. This expectation is not covered by ASTs created by JDT. For instance, an if-statement may have two children—a then and an else-part. Depending on the AST implementation, the access from the if-statement (parent) to the two parts (children) is not available through `getChildren()`, but through `getThenBlock()` and `getElseBlock()`.

Leaves in the intermediate AST are normal statements, with the statement kind as label and the statement itself as value. For instance, the leaf of statement `foo.bar();` has the label *MI* and the value *"foo.bar();"*.

5.2.3 Change Distilling Algorithm

The intermediate ASTs T_1 and T_2 are then fed into our change distilling algorithm (see Index 3 in Figure 5.3). The algorithm can be configured with different string and tree similarity algorithms and thresholds, as described in Section 4.2. The output is a set of basic tree edit operations that are classified to instances of change types (see Chapter 3).

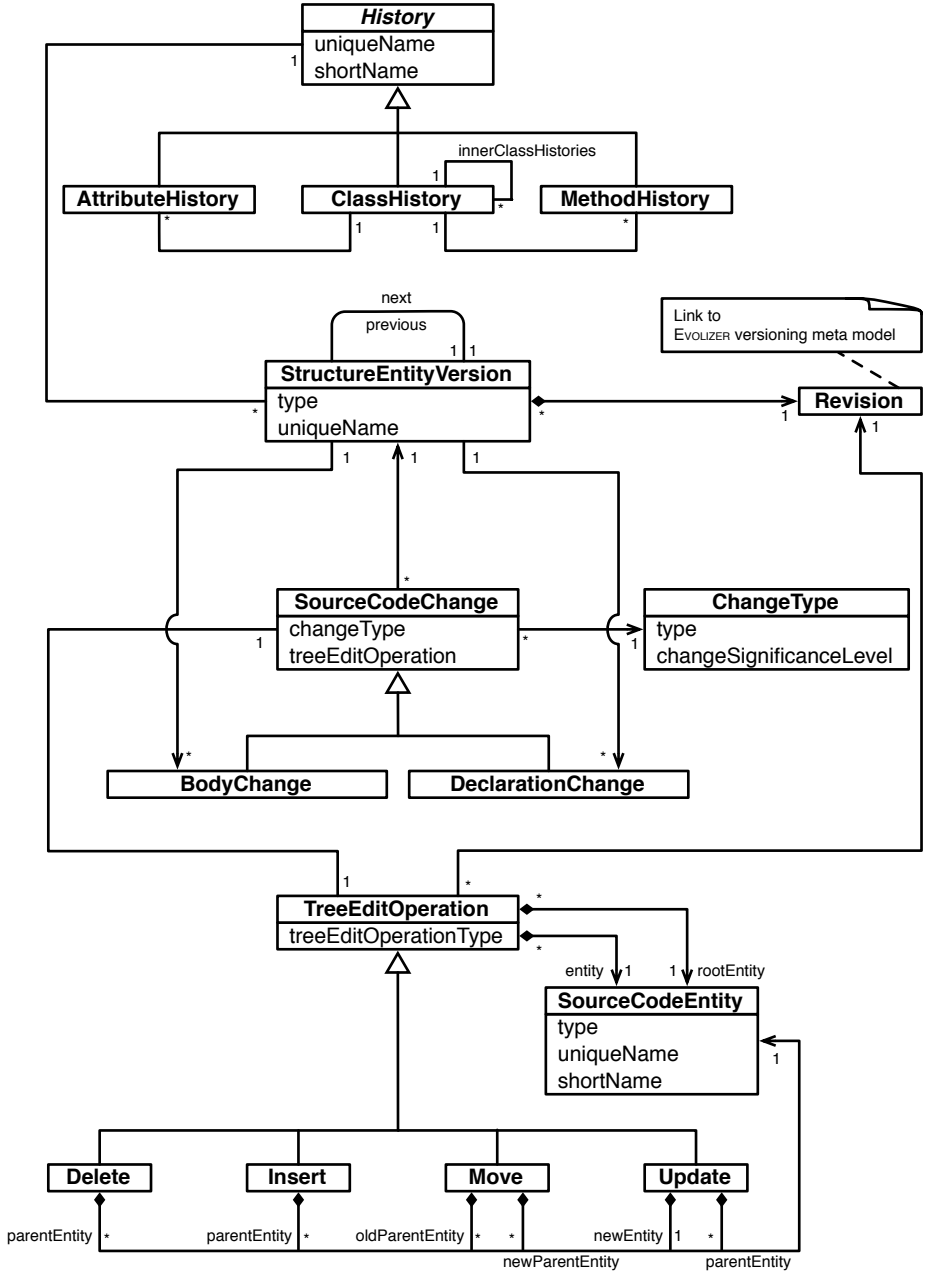


Figure 5.4: Change history meta model

Sometimes, we can infer that an update took place even if the similarity of the two strings under comparison is too low. Consider the methods `foo(Object myParam)` in revision $n-1$ and `foo(Figure myParam)` in revision n. A parameter type change from "Object" to "Figure" happened but the similarity of the two strings "Object" and "Figure" is below the threshold $f = 0.6$, hence is not matched. By classifying the tree edit operation without any further check, a new parameter would be inserted and an old one would be deleted. Since the parameter name did not change, the classifier is able to classify the two operations as a *Parameter Type Change* by checking whether the parents of "Object" and "Figure" are equal.

5.2.4 Change History Model Generation

When CHANGEDISTILLER extracts changes from the version history of a class, it builds a full instance of our change history meta model (see Index 4 in Figure 5.3). The change history meta model is based on the generic history model of Gîrba (2005; 2006) (see Figure 5.4, page 88). We explain the meta model with an example. After that, we describe the structure of the *source code changes* and *change types* in more detail. Figure 5.5 illustrates the construction of the change history model for the example. The names in `typewriter` font indicate the *unique names* that are assigned to the instances.

Assume CHANGEDISTILLER is extracting the change history of class `Example`. The class has three revisions: 1.1, 1.2, and 1.3. The method `exampleMethod(int)` is a member of the class and exists during the whole history. Another method, `addedMethod(long)` is added in Revision 1.3. The method `exampleMethod(int)` changes between Revisions 1.1 and 1.2 as well as between 1.2 and 1.3.

CHANGEDISTILLER starts with Revision 1.2. First, it creates an instance of `ClassHistory` for the class `Examples`. An instance of `MethodHistory` for the method `exampleMethod(int)` is also added because it has changed. The `uniqueName` of the method history is the fully qualified signature of the method. The changes between the method at Revision 1.1 and Revision 1.2 are extracted using our change distilling algorithm. For that, CHANGEDISTILLER distinguishes between `BodyChange` and `DeclarationChange`. Body changes are applied inside the method body, declaration changes on its declaration, *i.e.*, the signature as well as the return type and modifiers. CHANGEDISTILLER creates an instance of a `StructureEntityVersion` for the method version, links the instance to the Revision 1.2, and attaches the resulting changes including their `ChangeType` to the structure entity version. The `Revision` class is defined in the `org.evolizer.model.versioning` meta model (see Appendix C). The `StructureEntityVersion` instance is added to the method history. CHANGEDISTILLER does not add a `StructureEntityVersion` for the class because neither the class body nor its declaration changed. That the method changed does not count as a class body change.

Next, CHANGEDISTILLER processes Revision 1.3. First, it recognizes the insert

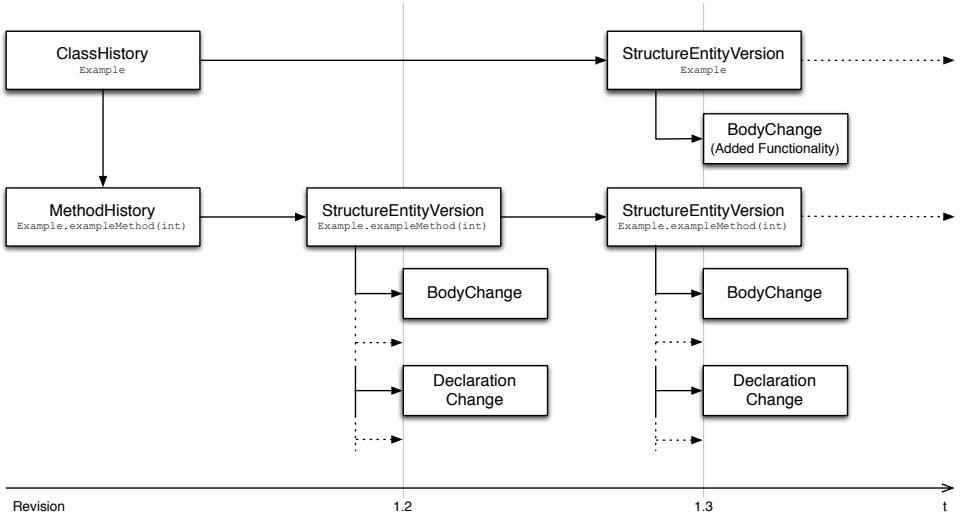


Figure 5.5: Generated change history model for the example. The time line indicates when the corresponding objects (instances) are created and added to the model.

of the method `addedMethod(long)`, creates a `StructureEntityVersion` instance for the class at Revision 1.3, and adds it to the class history. We defined the method insert change type as a change in the class body. A `BodyChange` with the corresponding `ChangeType` for the method insert is created and attached to the class version, *i.e.*, the corresponding `StructureEntityVersion` instance. CHANGEDISTILLER does not create a method history for the newly added method because did not change yet. The changes between the method `exampleMethod(int)` at Revision 1.2 and 1.3 are extracted and processed as in Revision 1.2. CHANGEDISTILLER has to decide whether a new method history starts or the method version can be added to an existing history. For that, it compares all signatures of existing method histories to the name of the method version. It uses a string similarity measure and a corresponding threshold (see Section 4.2). Either CHANGEDISTILLER finds a similar name and attaches it to the existing method history or it creates a new one. The unique name of the method history is adapted in cases the method signature changed.

Details of change types

A `SourceCodeChange` is the actual tree edit operation that reflects the change in the AST. It consists of a `ChangeType` that describes the change and a change significance level as we present in Chapter 3. The attribute `treeEditOperationType`

is either *insert*, *delete*, *move*, or *update*, *i.e.*, one of the basic tree edit operations. A subclass exists for each of those. A tree edit operation comprises of a set of `SourceCodeEntity` instances. For source code entities we use the terminology of the Java Development Tools[3] (JDT) of Eclipse. That means an AST node is represented by a source code entity in our meta model. The `type` of a source code entity is the AST node type defined in JDT, *i.e.*, the source code entity type.

Each source code change operates on at least two entities:

1. `rootEntity`: The root entity is the source code entity in which the change was applied. For instance, if a method invocation statement is inserted in a method, the method is the root entity.

2. `entity`: The entity is the source code entity on which the change is applied. For instance, if a method invocation statement is inserted or updated, the method invocation statement is the entity.

The `parentEntity` in `Insert`, `Delete`, and `Update` is the parent entity of the changed entity. For instance, if a method invocation statement is inserted into an if-statement, the if-statement is the parent entity.

The `oldParentEntity` and `newParentEntity` of `Move` described from where to where the entity is moved. For instance, if a method invocation statement is moved from a while-statement to an if-statement, the while-statement is the old entity and the if-statement is the new entity.

The `newEntity` of `Update` describes the entity after the update is applied. For instance, if the method invocation statement `foo.bar("string")` is changed to `foo.bar("newString")`, the statement `foo.bar("newString")` is the new entity.

Distinguishing body from declaration changes. The classes `BodyChange` and `DeclarationChange` are empty. This is a classic example of *taxomania* (Meyer, 1997). However, we still keep these classes because (1) we distinguish body from declaration change types explicitly and (2) storing the changes with Hibernate is more appropriate with separate subclasses.

5.3 Résumé

To analyze change types in real software systems we have integrated our change distilling algorithm into Eclipse as the CHANGEDISTILLER plugin. It applies tree differencing to subsequent ASTs of Java class revisions to extract the change types. Furthermore, during the extraction CHANGEDISTILLER builds a change history model and stores it along with the change types to our EVOLIZER RHDB. The plugin makes use of the compare and Java development functionalities in Eclipse.

[3] http://www.eclipse.org/jdt/

In this chapter we have shown that our change distilling algorithm can be implemented. We therefore accept Hypothesis H1a, and we regard Research Goal G2 as fulfilled.

III

Analyzing Change Types

6 Co-Evolution of Comments and Code

DOCUMENTING software is painful, especially when time is scarce and release deadlines are putting serious pressure on development teams, making it necessary to prioritize tasks. Under these circumstances feature implementation and bug fixing are focused most because customers usually pay for functionality in the first place and complain about non-functional requirements later on, when the impact of possible deficiencies in maintainability becomes apparent. The task of writing comments is often neglected by developers or given to staff members who are less familiar with the system, although every developer knows the value of good comments (Vanter, 2002).

Reading code is a fundamental task during software engineering (Goldberg, 1987)—code is read more often than it is written. Comments allow one to understand the code faster and deeper as well as to improve its readability (Spinellis, 2006; Tenny, 1988). They are crucial to sustaining software maintainability and aid in reverse engineering, for example, when applying the *Read All the Code in One Hour* reengineering pattern (Demeyer *et al.*, 2003). Elshoff and Marcotty (1982) stated that comments as well as the structure of the source code aid in program understanding and therefore reduce maintenance costs. Their finding was confirmed by the studies of Tenny (1988). Nonetheless, the example of Lakhotia (1993) showed that sometimes programmers do not care that someone else might want to understand the source code.

To understand whether the comments are a reason for decreasing maintainability in software projects, we study various productive software systems and address the following research questions in this chapter:

1. Is the growth factor the same for source code and comments, meaning that

about the same relative amount of code and comments is added over time? During the life cycle of a system, the API becomes more stable, most parts of the implementation have undergone several reviews, and re-documentation during maintenance takes places. We expect that the growth factor of code and comments approximate each other over time and keeps the ratio of commented source code stable.

2. Does the type of the source code entity have an influence on whether it gets commented and which source code entities are commented the most? The answer indicates whether developers are aware that commenting declaration parts and scopes increases readability and makes programs more comprehensible and therefore easier to maintain in the long-term.

3. Are comments adapted when source code is changed (*i.e.*, are comments kept up-to-date) and when does the adaptation take place—while changing the source code or in a later revision? By answering this question, we can draw conclusions on whether re-documentation is a integral part in the software engineering process, even tough programmers often neglect to adapt documentation to source code changes immediately.

To answer these questions we developed an approach to map comments to source code entities and to track co-changes of source code and comments over the history of a software system. We use the heuristics that the proximity between source code and comments indicate an association, and that comments describe the source code to which they are associated. To track co-changes we leverage data provided by our EVOLIZER and CHANGEDISTILLER.

When the process of associating comments to source code and extracting co-changes between them is completed we can answer questions like *"What is the most commented source code entity in a method body?"—e.g., "it is the if-statement,"* or *"When was the comment associated to a particular if-statement adapted to a condition change?"—e.g., "three revisions after the if-condition changed."*

For each research question we explain its rationale, define corresponding hypotheses, and conduct an empirical experiment with eight software systems. These studies consist of three major components of Eclipse, one commercial system, and four open-source systems from different domains. Based on the results of the experiments we statistically show:

1. The growth factor of source code and comments are similar over time in all investigated software systems. But this does not directly mean that newly added code is well commented because half of the investigated systems have a commented source code proportion of less than 50 percent. It rather means that the ratio of comments to source code remains stable.

2. The type of source code entity highly influences whether the entity is commented or not and there is also a partial order in the likeliness of whether

a certain entity gets commented. For example, if-statements are commented more often than simple statements.

3. Over 50 percent of comment changes are induced by source code changes. For six out of eight investigated systems over 90 percent of these co-changes are applied in the same revision.

The contributions of the this chapter are (1) an approach to map comments to source code entities, (2) an approach to track co-changes of source code and comments over the history of a software system, and (3) an empirical study on the commenting process in software systems. We also report on the experiences we have made when we applied our approach in industrial projects and provide a discussion on further applications of our work in terms of software quality analysis.

The remainder of this chapter is structured as follows. In Section 6.1 we present our approach to map comments to source code entities and to track co-changes. This approach is then applied to eight software systems and we discuss the results in Section 6.2. The interpretation in terms of software quality of the results is discussed in Section 6.3, including an assessment of our approach. We conclude this chapter with a reflection on the findings with respect to the hypotheses and research goals of this dissertation in Section 6.4.

6.1 Data Extraction and Collection

To answer our three research questions, we extract and collect data from three different sources. Counting the number of lines of non-commented source code and lines of comments is straightforward and not discussed in-depth. In this section we present our approach to map comments to arbitrary source code entities and to track co-changes among them.

Figure 6.1 gives an overview of the mapping, change detection, and co-change tracking process:

1. The source code of all revisions of a particular Java class is fetched from the EVOLIZER RHDB. Before using these revisions as input for CHANGEDISTILLER, we establish a mapping between comments and source code entities for each revision.

2. For each pair of subsequent revisions, we extract the change types of both the source code entities and the comments with our CHANGEDISTILLER, the implementation of our change distilling algorithm. The change types are stored in our EVOLIZER RHDB.

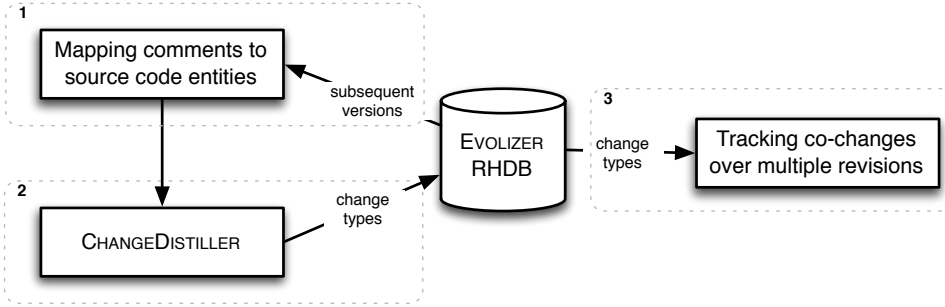


Figure 6.1: Overview on the change detection and tracking process

3. When this process is completed, a full-fledged change history is available for the class, allowing us to relate comment to source code changes and make a variety of observations, ranging from, *e.g.*, *"The most commented source code entity is..."* to more sophisticated ones such as *"The comment associated with a particular if-statement in method* `bar()` *was adapted three revisions after the condition of the if-statement had been updated."* By aggregating these observations we can especially analyse the process of adapting comments to source code changes of a software system.

6.1.1 Mapping Comments to Source Code Entities

In programming languages, it is seldom straight-forward to track relations between comments and source code entities algorithmically. Block and line comments cannot be assigned confidently to a particular adjacent entity by using purely syntactical rules. Because of that, Kaelbling (1988) proposed to remove line and block comments from programming languages and to introduce *scoped comments* instead. In today's programming languages, we still have to deal with line and block comments and, consequently, we have to establish a mapping by applying a set of heuristics.

We treat consecutive line comments as a syntactical alternative to block comments and merge them before we establish a mapping between source code entities and comments. Furthermore, we filter commented source code with a regular expression matcher that targets simple source code indicators such as patterns of parentheses, brackets, semicolons, and the like. We do not apply the matcher on API comments (*e.g.*, Javadoc) because they often contain code or code-like structures, such as source code snippets, giving an example on how a class or method is used properly. Another example are (semi-)formal specifications that define pre and postconditions. But we filter empty API comments, since IDEs may construct

empty API comments when a developer adds a new class, field, or method. Empty API comments are similar to:

```
/**
 *
 */
```

For each comment, we form triples of {preceding source code entity, comment, succeeding source code entity}. To find out whether a comment is associated with its preceding or succeeding source code entity, we apply the following set of heuristics on every triple:

- **Comment on the same line.** Comments and source code entities located on the same line are often associated. These kinds of comments clarify the meaning of the preceding source code entity, as shown in the following example:

```
int i = 0; // Iterator for while loop
```

- **Comment on an adjacent line.** Comments are normally in direct proximity of the corresponding source code entity. In the example below, each of the surrounding statements must be considered to be associated:

```
foo();
/* If foo() did not succeed,
   then calling bar() will
   raise an exception. */
bar();
```

- **Comment describes source code.** Each word appearing in the comment as well as in the source code entity is an indication that the comment belongs to the source code entity. We use a token-based measure (see Section 6.1.2 for details) to determine the similarity between comment and source code. We follow the heuristic that comments often pick up identifiers, *e.g.*, variable names, found in the code which they are describing. To separate tokens in comments and source code entities we use non-alphanumeric characters as delimiters. Concerning the example above, both method invocations, `foo()` and `bar()`, can be associated to the comment.

For both, the preceding and the succeeding source code entity, we compute a ranking based on these heuristics. We map the higher ranked entity to the comment. In the case that the ranking is even, the succeeding source code entity is chosen, since among developers, it is common practice to write comments preceding the associated source code statement or block. This was also confirmed during a discussion with Microsoft developers.

In the example above, all the heuristics apply on both source code entities `foo()` and `bar()`. They are adjacent to the comment in between them and have the same textual similarity—both words, "foo" and "bar," are in the comment. Because the ranking is even, we choose the succeeding entity, *i.e.*, `bar()`, as the associated entity.

6.1.2 Extracting Comment Changes

Source code changes are extracted and classified by our change distilling algorithm (see Chapter 4). The classification is based on our taxonomy of source code changes which defines source code change types according to tree edit operations in the abstract syntax tree (AST) (see Chapter 3). In particular, our change distilling algorithm applies tree differencing pairwise on subsequent versions of ASTs of classes to extract the tree edit operations. We have implemented the algorithm in the Eclipse plugin CHANGEDISTILLER (see Chapter 5) and it currently works with the Java programming language.

To extract comment changes with CHANGEDISTILLER, we have to match comment nodes in the ASTs across subsequent revisions. The matching between comments is computed by a token-based similarity measure. This takes comment updates also into account, which an exact matching would not detect. To compare two strings s_1 and s_2, the strings are first split into bags (multisets) of tokens, $T(s_1)$ and $T(s_2)$, according to a given non-alphanumeric separator.

The similarity value of the two strings is calculated as

$$sim(s_1, s_2) = \frac{|T(s_1) \cap T(s_2)|}{\max(|T(s_1)|, |T(s_2)|)}$$

Based on the evaluation with a test set we define that two comments c_1 and c_2 are similar if $sim(c_1, c_2) \geq 0.4$.

We use this similarity measure because it is robust to rearrangement of the text in a comment.

6.1.3 Relating Comment to Source Code Changes

Summarizing the steps described in the previous sections, we have gathered all data that we need to investigate whether or not comments are adapted when source code changes: (1) For each comment, we can compute to which source code entity it belongs, *i.e.*, which source code entity it describes; (2) the change types describe when and how source entities as well as comments have changed. By combining (1) and (2), we can address:

1. Whether a comment and its associated source code entity changed at the same time or the comment changed later,

2. Whether the changes were of the same type (insert, delete, move, or update), and

3. Which source code change type is most likely to induce a comment adaptation.

Consider the example chain of comment changes in Figure 6.2. In Revision 1.2, a comment, `/*threshold at 0.8*/`, is inserted for the source code entity (variable declaration) `double t = 0.8`. The source code entity changes in Revision 1.3, but the corresponding comment is not updated until Revision 1.4. Both, comment and associated source code entity, are deleted in Revision 1.5.

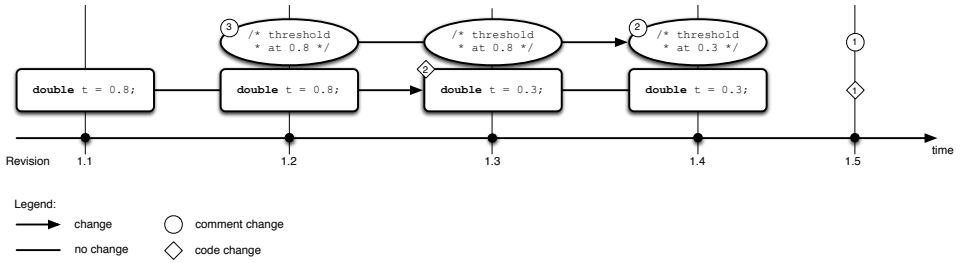


Figure 6.2: An example chain of comment changes. Co-changes of source code and comments have the same number

We reconstruct such chains backwards by starting with the latest revision r_i. Attached on each revision is a set of source code and comment changes C_i. For each comment change $cc \in C_i$ we check if the associated source code entity was also changed. If the associated entity changed as well, we stop and store that there was a *co-change* between the comment and its associated entity, whether they changed the same way (*i.e.*, insert, delete, move, or update), and the change type of the associated entity. In our example (Figure 6.2), we start with the comment deletion in Revision 1.5. The associated entity and the comment changed in the same revision and in the same way.

If the associated source code entity did not change in r_i, we check for corresponding changes in r_{i-1}, thus go backwards. This step is repeated until we either find a change of the associated entity, or another change of the comment. In the former case, we store that there was a *shifted co-change* between the comment and its associated entity. If another change of the comment occurs, a new element in the chain begins, and we state that cc occurred without a source code change. In

our example the comment in Revision 1.4 was changed one revision later than its associated entity. The comment insert in Revision 1.2 happened without a corresponding source code change.

The investigation of our example chain answers the third research question we posed at the beginning of this chapter and its results can be summarized as follows: The comment changed three times. The last change (in Revision 1.5) happened in the same revision accompanied by a change of the associated entity of the comment. They had a co-change and both, the comment and the entity, changed the same way (delete). The second change (in Revision 1.4) occurred one revision later than the change of its associated entity, thus, they had a shifted co-change. The first comment change (in Revision 1.2) was applied solely. We can also state that it is more likely that a statement delete induces a comment change in the same revision than a statement update does.

We also check whether a comment describing a scope has changed due to source code changes inside the scope. For instance, when the body of an if-statement changed, it is likely that the comment describing the if-statement changed as well. But so far we have still open issues on this concern: We are unable to extract such shifted co-changes. Our change extraction model stores source code entities only when they change. But to reconstruct co-changes over scopes, a complete source code model with unique identifiers is necessary. Since organizing and storing all this data suffers from a remarkable performance and storage overhead, we decided to skip this data. For co-changes in the same version, we can leverage the information on source code location of the entities and comments to reconstruct the scoping.

To distinguish comment changes that are induced by a change of its associated entity and comment changes that are induced by changes inside the scope of its associated entity, we speak of *direct co-changes* and *scope co-changes*.

6.2 Empirical Results

In this section we describe our results of applying our comment to source code mapping and co-change tracking approach. In Section 6.2.1 we present the experimental setup; in Section 6.2.2 we validate our data extraction and collection process; in Sections 6.2.3–6.2.6 we evaluate the results and describe our findings.

6.2.1 Experimental Setup

We conducted an empirical study with eight case studies. These studies consist of three major components of Eclipse, one commercial system, and four open-source systems from different domains:

1. ArgoUML (UML designing tool; observation period: Jan 98 – Dec 05)
2. Azureus (Java bittorrent client; observation period: Jul 03 – May 07)
3. Eclipse Core (21 plugins from the Eclipse platform component; observation period: May 01 – Sep 07)
4. Eclipse JDT (17 plugins from the Java Development Tools component; observation period: May 01 – Sep 07)
5. Eclipse PDE (5 plugins from the Plugin Development Environment component; observation period: May 01 – Sep 07)
6. jEdit (text editor; observation period: Sep 01 – Jun 06)
7. JFreeChart (Java chart library; observation period: Oct 01 – Jul 07)
8. Webframework (a commercial framework for web applications; observation period: Jul 04 – Sep 07)

All projects are written in Java and are version-controlled using CVS. ArgoUML, jEdit, and JFreeChart have already moved to Subversion. For jEdit we received an older repository directly from the developers, for ArgoUML we use the repository provided by the MSR Workshop Challenge of 2006, and for JFreeChart, the CVS repository is still available on sourceforge.net. Table 6.1 summarizes the software systems.

In the remainder of this section, we validate the parts of our data extraction and collection process (Section 6.2.2). Then, we explain the underlying rationale, formulate the hypotheses, perform a corresponding experiment on the case studies and discuss the results for each research question (Sections 6.2.4–6.2.6).

6.2.2 Validation of Data Extraction and Collection

For each step of the data collection process we validated its output to show the accuracy of the process. In this section we focus our validation on filtering commented source code as well as on the mapping between source code entity and comments. The change extraction validation is described in-depth in Chapter 4.

Comment filtering

We have randomly selected 8,978 comments from the latest releases of the eight software systems and inspected them manually to decide whether they are comments or commented source code. Out of these comments, we classified 372 as commented source code. Our simple pattern matching algorithm found 240 true positives, 87 false positives, and 132 false negatives leading to a precision of 0.73

System (# releases)	# source revisions	# changes	# comment changes (%)	LOC first	LOC last
ArgoUML (14)	39,421	183,752	24,049 (13%)	200,735	239,791
Azureus (12)	33,008	245,214	13,790 (6%)	17,227	362,316
Eclipse Core (15)	15,454	69,383	9,714 (14%)	61,592	133,574
Eclipse JDT (15)	121,442	904,786	79,351 (9%)	420,233	974,006
Eclipse PDE (15)	35,137	153,891	6,534 (4%)	66,638	225,516
jEdit (12)	6,754	88,932	8,887 (10%)	80,726	133,895
JFreeChart (10)	4,675	23,678	3,166 (13%)	151,040	250,180
Webframework (13)	19,501	116,994	9,735 (8%)	43,452	124,796
Total	275,392	1,786,630	115,226 (9%)	1,041,643	2,444,074

Table 6.1: Analyzed software systems. The number in parentheses beside the system name indicates the number of releases we investigate. [# source revisions] indicates the total number of revisions of Java files. [# changes] indicates the number of changes type occurrences that were applied during the period. [# comment changes] indicates the number of comment change type occurrences that were applied during the period. [LOC first] and [LOC last] indicate the lines of code for the first and the last release of the component in the period

and a recall of 0.65. Since simple regular expressions do not have the power of a parser but show a better runtime performance, these numbers of false positives and negatives are acceptable. Nevertheless, we expected to gain a higher recall. We found that 88 (66%) of the false negatives are due to comments at the beginning of files in JFreeChart. Our regular expression matcher filtered them as commented source code because such comments include code characters, such as '`=`' or '`]`', as delimiters and dotted expressions, such as `1.2.3`.

Comment to source code mapping

We checked whether comments are mapped to source code, but we did not validate whether the association is semantically correct. This validation issue is discussed in Section 6.3.

We have randomly selected 761 comment and source code mapping pairs. The manual inspection of these pairs gave a precision of 1.0 (0 false positives). By randomly selecting mapping pairs, we are not able to collect the false negatives because they are not in the set of mapping pairs. By counting the number of unmapped comments, we collected the false negatives for the whole data set. In total 258,555 comments were extracted from the eight software systems but 7,682 (3%) could not be mapped (false negatives). Over half of the false negatives (62%) are found in jEdit. Developers of jEdit block any type of scope (classes, methods, if-statements, etc.) with beginning and ending line comments: `//{{{ Debugging`

and `//}}}`. Our algorithm deals with triples of {source code entity, comment, source code entity} to decide whether a comment belongs to its preceding or succeeding entity. As such triples are missing at multi-level scope ends, the special jEdit comments are not mapped. Although the impact of this limitation is tremendous in jEdit, we decided not to overcome this situation in general because jEdit is an outlier compared to the other investigated systems. Removing jEdit from the data set leads to 253,778 comments in total, of which 2,905 (1%) could not be mapped.

6.2.3 Organization of Experiments

We organize our experiments and discussion of the results according to the scheme of Baresi and Morasca (2007):

Question. This is the underlying research question that we want to answer.

Rationale. The reason why we claim that the research question is relevant.

Hypothesis. We outline the claim whose truth we want to check with our empirical analysis and describe the statistical hypothesis that we test.

Results. We present the results we obtain from the empirical studies and how we test the hypothesis.

Discussion. We discuss the results and reflect how the observations relate to the research question.

Summary of experiment. We give a summary about the findings of the experiment.

6.2.4 Experiment 1: Growth Factors of Code and Comments

Question. Is the growth factor the same for source code and comments, meaning that about the same relative amount of code and comments is added over time?

Rationale. Answering the first research question allows us to better understand the life cycle of software systems. Intuitively, software systems tend to become more mature with every release: The public API becomes more stable, most parts of the implementation have undergone several reviews, and re-documentation during maintenance takes places. We expect that the growth factor of code and

comments approximate each other over time and keeps the ratio of commented source code stable.

Hypothesis. We expect that the growth factor is the same for source code and comments. Let R_i be a release and R_j its succeeding release of a software system. $g_{comment_{ij}}$ is the growth factor of comments and $g_{code_{ij}}$ is the growth factor of source code between the releases R_i and R_j. If comments and source grow in the same proportion the difference $d_{ij} = g_{comment_{ij}} - g_{code_{ij}} = 0$. We formulate the following hypothesis to express our assumption of equality in growth between source and comment: The difference d_{ij} between any pair of subsequent releases R_i and R_j of a software system equals 0.

Results. We decided to use a *two-tailed one-sample t-test* to statistically verify if our hypothesis holds or can be rejected. The *t-test* is adequate as significance test in our case because the variance σ is unknown and the size n of the sample set is < 30. Considering our hypothesis the expected value of d_{ij} is 0. We then have a *Student's t-distribution* with the parameter $\mu_0 = 0$. The test was performed under exact the same setup for every software system shown in Table 6.1: We first extracted the set A of all $(g_{comment_{ij}}, g_{code_{ij}})$ pairs for any two subsequent releases R_i and R_j. We then calculated the set B of all differences d_{ij} for every $(g_{comment_{ij}}, g_{code_{ij}})$ pair in A and calculated $\bar{x}$ as the arithmetic mean for all d_{ij} in A. This leads to the equation were the *t-value* is defined as (Dowdy *et al.*, 2004)

$$t = \frac{\bar{x} - 0}{s} \cdot \sqrt{n}$$

and n is the size of the set B referring to the number of calculated differences d_{ij}. s is the standard deviation, *i.e.*, the square root of the experimental sample variance calculated on B. We tested t against a significance level $\alpha = 0.01$ resulting in a rejection region $t > t_{0.995,\nu}$, where $\nu = n - 1$ describes the *degrees of freedom* parameter of the *Student's t-distribution*. In Table 6.2 we list the numbers necessary for the t-test and show that our hypothesis is accepted in all cases.

Discussion. Figure 6.3 depicts the results of Experiment 1. For each system we plotted the number of non-commented lines (NCLOC) of code and number of comment lines (NCL) with a solid line. The dashed lines are the growth factor of NCLOC (GF-NCLOC) and the growth factor of NCL (GF-NCL). The plots show the acceptance of our hypothesis. The course of the dashed lines mostly coincide. Although, both, source code and comments, grow equally in all investigated software systems, newly added code is barely commented in half of the systems: Azureus,

System	$\bar{x}$	s^2	n	t	Accept ($\alpha = 0.01$)
ArgoUML	0.058	0.0104	13	2.04	yes
Azureus	0.143	0.2789	11	0.90	yes
Eclipse Core	0.012	0.0014	14	1.24	yes
Eclipse JDT	-0.002	0.0007	14	-0.30	yes
Eclipse PDE	0.012	0.0014	14	1.24	yes
jEdit	0.014	0.0005	11	2.18	yes
JFreeChart	-0.007	0.0007	9	-0.84	yes
Webframework	-0.027	0.009	12	-0.98	yes

Table 6.2: Data of Experiment 1. $\bar{x}$ indicates the arithmetic mean for all d_{ij}. s^2 indicates the sample variance. n indicates the number of calculated differences d_{ij}. t indicates the calculated t-value

Eclipse JDT, Eclipse PDE, and jEdit have only between 27% (Eclipse PDE) and 42% (jEdit) commented source code. In contrast, the systems ArgoUML, Eclipse Core, JFreeChart, and Webframework have between 53% (Webframework) and 100% (JFreeChart) commented lines of source code. Except for ArgoUML and Azureus, this commenting process is constant for all systems. For ArgoUML the commenting process was getting better, meaning that newly added code was commented more intensely after Release 0.15.6. Figure 6.3 depicts that for the Releases 0.15.6 and 0.16.1 of ArgoUML the growth factor of NCL is bigger than of NCLOC. The commenting process of Azureus is the opposite; after Release 2.1.0, source code was getting barely comment.

Since the core Eclipse IDE is mainly developed at IBM and because they have coding conventions that are valid for all Eclipse components, the differences between the commenting process of the three Eclipse components is surprising. The only component that has a high commented source code ratio is Eclipse Core—90% on average. Eclipse JDT has a ratio of 40% and Eclipse PDE of 24% in average. A possible explanation for these discrepancies is that the ratio between public API and internal implementation in Eclipse Core is higher than in JDT and Core. Eclipse is known to have a comprehensive public API documentation, but a modest internal implementation documentation, as confirmed by Schreck *et al.* (2007). The second experiment shows which type of source code is more likely to be commented. This will explain the differences in the commenting process of the three Eclipse components.

The reason for the high percentage of commented source code in JFreeChart are the long file header comments, but also the intensive API documentation—an exemplar of well documented software.

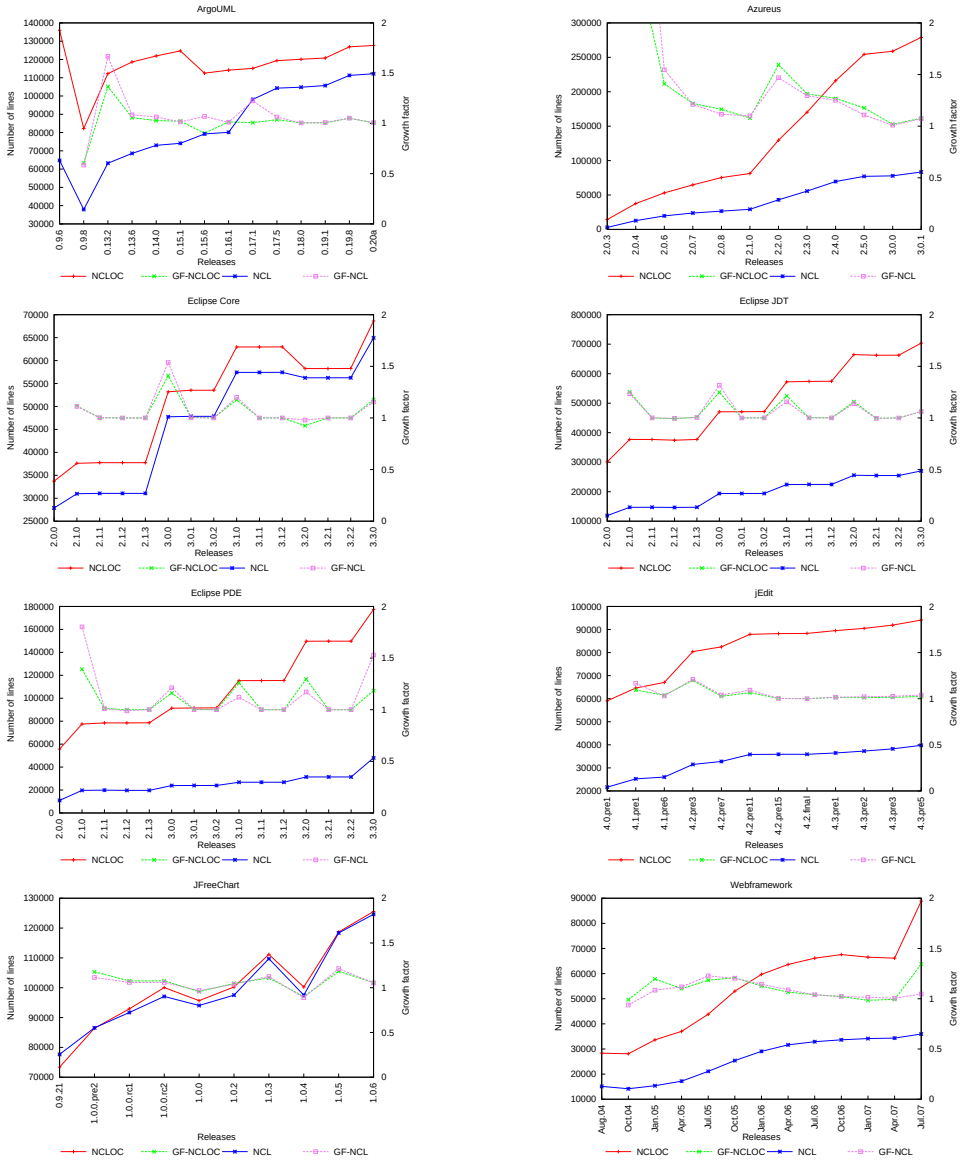


Figure 6.3: Results of Experiment 1. NCLOC and NCL indicate the number of non-commented lines of code and number of comment lines. GF-NCLOC and GF-NCL indicate the growth factor of NCLOC and NCL between two subsequent releases. The growth factor curves are dashed

Summary of Experiment 1. We have statistically shown that source code and comments grow equivalently over time in all eight software systems. We have also shown that this does not directly mean that newly added code is well commented in all systems. Half of the investigated systems have a commented source code ratio of less than 50%. Even systems that are developed in the same company, such as IBM for Eclipse, have different commenting characteristics. To conclude, equal growth factors are an indication that the ratio of commented source code remains stable. But, it does not mean that newly added code is commented comprehensively.

6.2.5 Experiment 2: Commented Source Code Entities

Question. Does the type of the source code entity have an influence on whether it gets commented and which source code entities are most likely to be commented?

Rationale. Do all source code entities have the same likelihood for being commented? Or more precisely, is there any statistical evidence that programmers are more likely to add documentation to building blocks of a program, such as class or method declarations, or to if and loop scopes, rather than to simple statements? These questions are relevant because the answer indicates whether developers are aware that commenting declaration parts and scopes increases readability and makes programs more comprehensible and therefore easier to maintain in the long-term.

Hypothesis. We claim that the type of a source code entity has an influence on whether it gets commented or not. The statistical hypothesis we test is, whether source code is commented or not is independent from the source code entity type.

Results. We decided to use the *two-variable* χ^2*-test* to statistically verify our hypothesis. The χ^2*-test* evaluates whether observed frequencies reflect the independence of two qualitative variables. In our data set, the first variable describes whether a source code entity type is commented or not. The second variable describes the source code entity type.

For each software system shown in Table 6.1 we calculated the (observed) numbers of commented as well as non-commented source code entity types and the corresponding expected values of the latest release. This calculation results in contingency tables as shown in Table 6.3 for ArgoUML. The contingency tables for all other systems are listed in Appendix D.

Obs.	Class	Field	Method	If	Loop	VD	Call	Other	Total
c	1,659	1,606	12,347	765	81	1,034	744	595	18,831
$\bar{c}$	33	1,852	0	9,469	1,531	10,577	17,255	42,832	83,414
Total	1,692	3,458	12,212	10,234	1,612	11,611	17,999	43,427	102,245

Exp.	Class	Field	Method	If	Loop	VD	Call	Other
c	312	637	2,249	1,885	297	2,138	3,315	7,998
$\bar{c}$	1,380	2,821	9,963	8,349	1,315	9,473	14,684	35,429

Table 6.3: Observed (Obs.) and expected (Exp.) contingency tables of Release 0.20a of ArgoUML. c indicates the number of commented source code entity types. $\bar{c}$ indicates the number of non-commented source code entity types. The values in the expected contingency tables are rounded. VD means variable declaration

Expected values of contingency tables are calculated as follows: The expected value e_{ij} in the ijth cell is defined as in (Dowdy *et al.*, 2004):

$$e_{ij} = \frac{(o_{i.}) \cdot (o_{.j})}{o_{..}}, \quad \text{where } o_{i.} = \sum_j o_{ij}, \quad o_{.j} = \sum_i o_{ij}, \quad o_{..} = \sum_i \sum_j o_{ij}$$

where o_{ij} is the observed number in the ijth cell of the contingency table. The χ^2 value is then defined as

$$\chi^2 = \sum_i \sum_j \frac{(o_{ij} - e_{ij})^2}{e_{ij}}$$

We tested χ^2 against a significance level $\alpha = 0.005$ resulting in a rejection region $\chi^2 > \chi^2_{0.995,\nu=7}$, where $\nu = (r-1) \cdot (c-1)$ is the degree of freedom, a function of the number of rows, r, and the number of columns, c, of the contingency table.

The χ^2 values of the eight software systems are all >11,600. Since $\chi^2_{0.995,7} = 20.278$ the hypothesis is rejected and we conclude that whether a source code entity gets commented or not depends on its type.

Discussion. The results of the statistical tests show that the source code entity types do not have the same likeliness for being commented in all investigated software systems. We expected this result because commenting high level scopes, such as methods or classes, has a higher impact for understanding software than lower level scopes or simple statements.

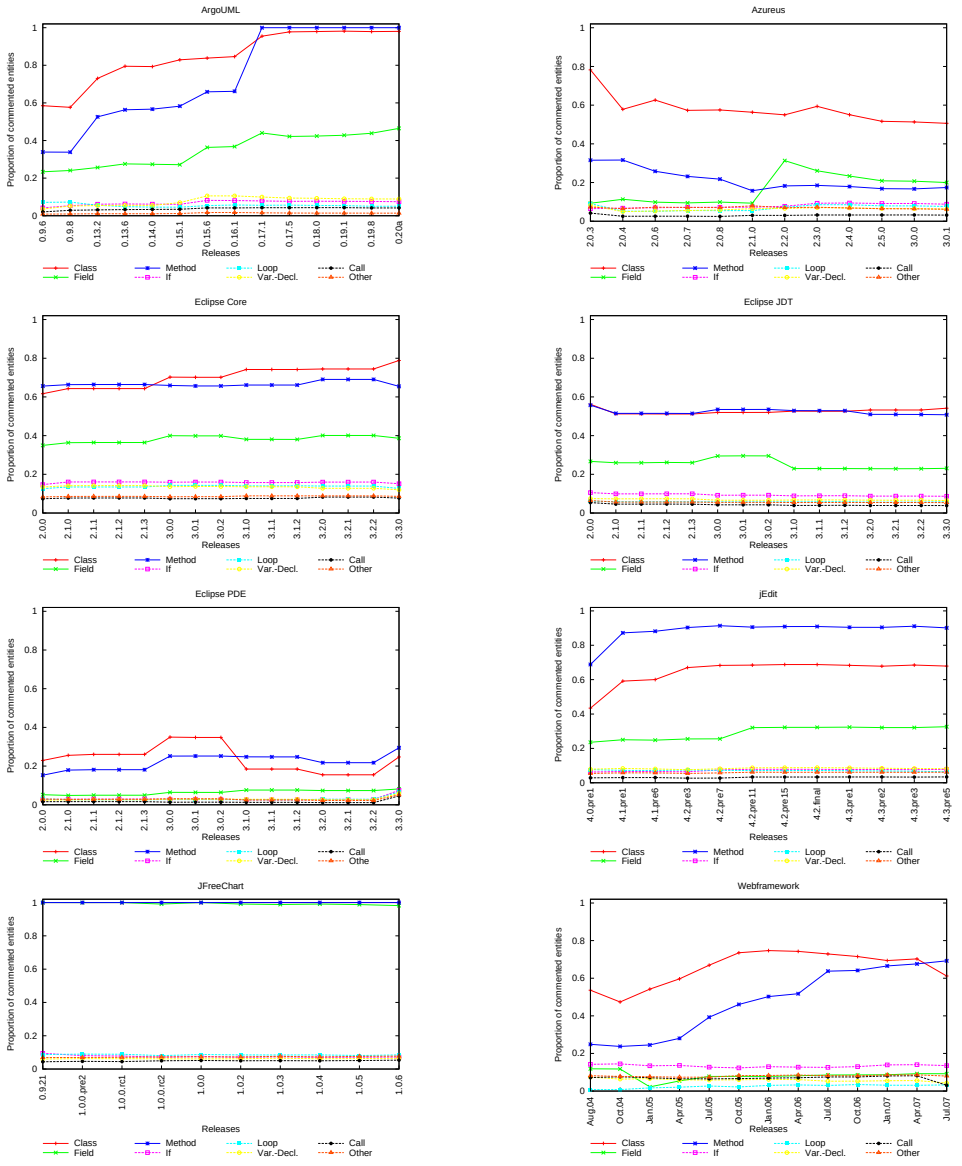


Figure 6.4: Results of Experiment 2. For each release and source code entity type of the investigated software systems we plot the proportion of commented source code entity types. The curves of the high-level constructs (class, method, and field) are solid, those on statement levels (if-statement, loop-statement, variable declaration, call, and other) are dashed

Figure 6.4 depicts the results of Experiment 2. According to the diagrams in this figure, there is a partial order in the likeliness whether a certain type of source code entity gets commented or not. The high level scopes, class, method, and field, are more commented than the low level scopes or simple statements.

Except for jEdit and JFreeChart, the order of high level scopes changed over time. In jEdit methods are more commented than classes and fields. In JFreeChart each class, method, and field was commented in all releases. JFreeChart is an exemplar for well documented API. In Azureus and Eclipse PDE commenting the API was neglected since the beginning of the project. The percentage of commented classes decreased in both systems towards the latest release; in Azureus the same characteristics applies to the methods as well. All other systems either have a stable or increasing percentage.

Low level scopes and simple statements are barely commented in all systems; the highest values are found in Eclipse Core (<16%). In all other systems they are below 10%. The order of commenting low level scopes and simple statements varies for the eight systems. If-statements are commented most frequently for five of them; in ArgoUML and jEdit the most often commented low level entity type are variable declaration statements, in JFreeChart the loop statements. Except for ArgoUML, calls are the least commented low level entity types.

The three diagrams of the Eclipse components show the reason why the ratio of commented source code is higher in Eclipse Core than in Eclipse JDT and PDE. In Eclipse Core classes and methods are about 20% more often commented than in Eclipse JDT and about 45% more often than in Eclipse PDE. Low level scopes and simple statements are also more often commented in Eclipse Core than in JDT or PDE, but not to that extent as high level scopes.

Overall, the only system that consistently comments high level scopes is JFreeChart. ArgoUML and partly jEdit have at least high commenting coverages for methods and classes. For all other systems, these coverages hardly go over 80%. One of the reasons for the low high level scopes commenting is that often private members are not commented.

Summary of Experiment 2. We have statistically shown that whether a source code entity gets commented or not depends on the type of entity. We have also shown that there is a partial order in the likeliness of whether a certain source code entity gets commented. High level scopes, such as classes, methods, or fields are more likely to be commented than low level scopes and simple statements, such as if-statements or calls. During the history of five of the investigated software systems the commenting percentages stayed stable, in two they increased and in one they partly decreased. A consistent commenting process is only identifiable in JFreeChart. Although it is well-known that comments describing low level scopes help in understanding them, they are barely commented in all investigated systems.

6.2.6 Experiment 3: Co-Change of Comments and Code

Question. Are comments adapted when source code is changed (*i.e.*, are comments kept up-to-date) and when does the adaptation take place—while changing the source code or in a later revision?

Rationale. By answering this question, we can draw conclusions on whether redocumentation is an integral part in the software engineering process, even though programmers often neglect to adapt documentation to source code changes immediately. In other words, we want to analyze if development in general follows a cycle similar to *apply bug or feature request* $\rightarrow$ *commit source code changes* $\rightarrow$ *adapt documentation* $\rightarrow$ *commit comment changes* $\rightarrow \cdots$?

Hypothesis. To keep comments up-to-date, we expect that the majority (*i.e.*, >50%) of the comment changes were induced by changes of their associated source code entity in the same revision.[1]

Results. To answer the third question, we have calculated chains of comment changes for each system. The results can be found in Table 6.4. The column "co-change" includes both, co-changes and shifted co-changes. To recapitulate, we speak of a *direct* (shifted) co-change if the comment change is induced by a change of its associated source code entity; and we speak of *scope* co-change if the comment change is induced by a source code change inside the scope of its associated entity. Scope co-changes include scoped statements as well as declarations whereas shifted scope co-changes only include declarations.

The hypothesis is accepted for a software system, if the "co-change" column multiplied with the "$\triangle_r = 0$, both" column is >50%. This multiplication indicates the percentages of comment changes that were induced by changes of their associated entity in the same revision. As the results in Table 6.4 show, we can accept the hypothesis for four software systems: Azureus, Eclipse JDT, Eclipse PDE, and JFreeChart. For the other four systems less than 50% of comment changes were induced by changes of their associated entity in the same revision.

Discussion. There is a significant difference in the behavior of direct and scope co-changes. During the evolution of all systems, 98% of direct co-changes happen in the same revision; there are only few shifted direct co-changes. In contrast to the direct co-changes, between 57% (Eclipse Core) and 93% (jEdit) of scope co-changes happen in the same revision. We can also observe that shifted co-changes between

[1] Compared to the hypotheses of Experiment 1 and 2 this is rather an assumption than a statistical hypothesis. Nevertheless we use the term hypothesis to keep the organization of the experiments consistent.

		$\triangle_r = 0$			$\triangle_r = 1$		
System	co-change	both	direct	scope	both	direct	scope
ArgoUML	50.66	81.58	98.14	62.55	4.69	0.22	9.87
Azureus	68.80	97.91	98.77	90.77	0.51	0.29	3.05
Eclipse Core	53.26	89.23	99.02	57.23	2.53	0.10	10.49
Eclipse JDT	65.93	93.70	98.83	84.95	1.31	0.18	3.39
Eclipse PDE	61.03	93.44	98.66	66.15	1.51	0.89	7.87
jEdit	50.06	96.67	98.85	92.98	0.54	0.29	1.02
JFreeChart	56.47	91.49	99.49	75.26	4.20	0.17	12.37
Webfrmwk	52.73	91.31	98.76	72.93	1.83	0.16	6.15

	$\triangle_r = 2$			$\triangle_r = 3$			$\triangle_r > 3$		
System	both	direct	scope	both	direct	scope	both	direct	scope
ArgoUML	2.94	0.16	6.13	1.83	0.23	3.66	8.97	1.26	17.79
Azureus	0.35	0.16	1.74	0.20	0.07	1.13	1.03	0.72	3.31
Eclipse Core	1.28	0.10	5.12	1.16	0.20	4.29	5.80	0.58	22.87
Eclipse JDT	0.74	0.06	1.86	0.58	0.05	1.47	3.66	0.87	8.32
Eclipse PDE	1.11	0.12	5.62	1.31	0.06	7.02	2.62	0.27	13.34
jEdit	0.36	0.25	0.54	0.34	0.14	0.66	2.10	0.47	4.80
JFreeChart	1.76	0.17	4.98	1.08	0.00	3.26	1.47	0.17	4.12
Webfrmwk	1.89	0.05	6.35	0.94	0.19	2.74	4.03	0.82	11.83

Table 6.4: Data of Experiment 3. [Co-change] indicates the proportion of comment changes that are induced by source code changes in the same revision or later. $\triangle_r$ indicates the number of revisions that elapsed between the source code and the comment change. We distinguish between direct and scope co-changes. [Both] merges [direct] and [scope]. The values in the table are in %. The percentages of these columns are relative to the "co-change" column. For instance, for ArgoUML 81.58% of all co-changes happen in the same revision

API comment and declarations occur: In Eclipse Core, for instance, in 10% of the scope co-changes the comment changed one revision after the scope changed. And, in 23% of the scope co-changes the comment changed more than three revisions after the scope changed. Systems that have shifted scope co-changes either have a significant amount of shifted co-changes in $\triangle_r = 1$ or $\triangle_r > 3$. That means, the comment is either adapted shortly after the code is changed or long after the code has changed; for instance during a consolidation phase before a release.

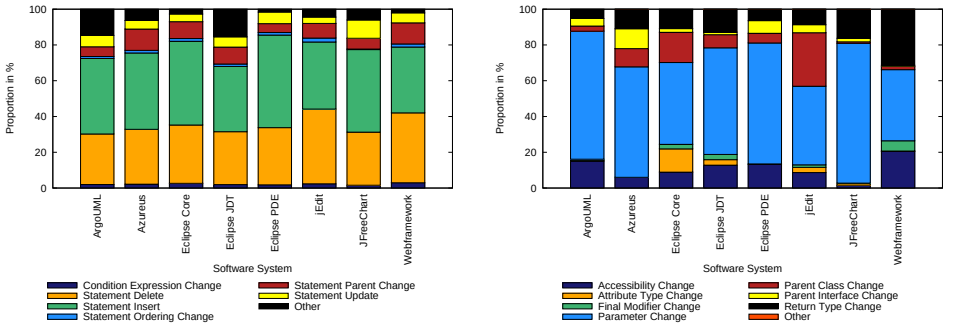
For all direct co-changes and direct shifted co-changes we have calculated the proportions of source code change types that induce comment changes. We distinguish between change types in the method body (body change types) as well as class, attribute, and method declaration change types (declaration change types).

These proportions are listed in Table 6.5.

Project	body	declaration	Project	body	declaration
ArgoUML	72%	28%	Eclipse PDE	87%	13%
Azureus	92%	8%	jEdit	77%	23%
Eclipse Core	76%	24%	JFreeChart	32%	68%
Eclipse JDT	81%	19%	Webframework	66%	34%

Table 6.5: The proportions of body and declaration change types that were responsible for direct and direct shifted co-changes

The proportion of body to declaration change types that induced comment changes are related to the results of Experiment 2. The more classes, fields, and methods are commented the higher is the proportion of declaration change types that induce comment changes. JFreeChart and Webframework have the highest proportion of declaration change types inducing comment changes, whereas comment changes in Azureus or Eclipse PDE are induced mostly by body change types.



(a) (b)

Figure 6.5: Distribution of body (a) and declaration change types (b) that induce comment changes

Figure 6.5 shows the distribution of body and declaration change types that induced comment changes. In all software systems the change types *statement insert* and *statement delete* induced the most comment changes. It is not surprising that statement delete has such an influence on comment changes because when a developer deletes a statement, she can delete the corresponding comment right away. Statement insert is one of the most applied change types and, therefore, it is

obvious that it induces most of the comment changes. The change types *statement parent change* and *statement update* cause also comment changes. It is surprising that statement updates have a marginal influence on comment changes. We expected a stronger relation because statement updates are applied often, and after such a change a comment might be outdated. A similar argument is valid for the surprising observation that *condition expression changes* do not have a significant influence on comment changes. The change type *other* means that we still encountered co-changes between comments. As we explain in Section 6.3, we still have issues with successive comments that are in different scopes.

Parameter changes are responsible for the most API comment changes in all investigated software systems. This is obvious because parameters are mostly described in API comments with a corresponding tag. In addition, IDEs, such as Eclipse, support the adaptation of such tags when declarations are changed. *Return type changes* influence changes in the API comment because they also have a predefined tag. A reason that *accessibility changes* have a significant influence on API comment changes may be as follows. Assume the modifier `private` of a method is changed into `public`. Because of that, either a API comment has to be added to describe the method, or the API comment has to be complemented with more detailed information. Changing the parent class or parent interface of a class has partly an influence on changing API comments.

Summary of Experiment 3. Over 50% of the comment changes are induced by source code changes. We have shown that direct co-changes happen in over 98% of the cases in the same revision. But we observed that API comments are more often adapted retroactively than other comments. This seems reasonable under the assumption that there are often public interfaces involved, which are more likely to be subject of re-documentation.

Source code change types that induce comment changes can be split into body and declaration change types. Statement insert and delete are the body change types that induce the most comment changes. Parameter changes, return type changes, and parent class or parent interface changes are the declaration change types that induce the most API comment changes.

6.3 Discussion

Our investigations of comments and their changes exposed interesting insights of the commenting process of software systems. In this section, we report on how we can leverage these investigations in terms of software quality. We also discuss whether our comment to source code mapping approach is appropriate.

6.3.1 Interpretation in Terms of Software Quality

Comments describe the source code of a software system. If they exist and are meaningful, they can aid in comprehending the system. In addition, meaningful comments allow us to reason about the source code and aid assessing its quality. Based on our investigations of comments and their change process we contribute to the quality assessment of comments. With our approach we can assess comments quantitatively and reflect on the commenting process of software systems. The results of our analysis can be compared against those for other projects and serve as an assessment for a particular aspect of the quality of a software and its development process.

For example, two of our industrial partner asked us to perform a quality analysis of their software systems. One of them was the company developing the Webframework.[2] Among other investigations, we suggested to analyze the commenting process of their projects and successfully applied the corresponding quality assessment. We briefly report on these experiences as well.

Assessing comments quantitatively

Experiments 1 and 2 assess the quality of the comments on a quantitative basis.

Experiment 1. Comparing the growth factor of the number of comment lines and the number of non-commented lines of code shows whether the proportion of comment lines to code lines increased, decreased, or stayed stable over the history of a software system. This neither indicates that the source code is well commented nor that the comments are meaningful. But it shows whether or not developers of a system comment their code consistently over time.

Experiment 2. The results of Experiment 1 give an impression on the amount of comments in a software system. Conducting this experiment is straight-forward and can be done with modern IDEs on the fly. But, simply counting the lines of code and the comment lines hides two major aspects of commenting: First, dead code is counted as comment lines. Second, which source code entity types are commented is not considered. We complement the interpretation of the results of Experiment 1 with Experiment 2 because dead code harms the comprehension of source code, and it makes a difference which and to what extent a certain source code entity type is commented to measure the quantitative quality of the comments. The less dead code is present and the more declaration parts as well as scopes are commented, the better the quantitative quality of the comments and the higher the maturity of the system.

[2] The detailed results of the other study are not available for publication.

Experiences with industrial partners. We have accompanied the development of the Webframework since April 2005 and periodically assessed the quality of the source code. At the beginning of this evaluation, we regarded the overall percentage of commented source code as sufficient but suggested to improve the quantity of API comments. The company agreed on this quality factor and increased the proportion of commented methods and classes as Figure 6.4 shows—our toolset allowed us to quickly assess the improvements quantitatively.

We also applied our investigations on the second commercial software system. The results of Experiment 1 gave the impression of a sufficient commented system. But the Experiment 2 showed that a lot of dead code was present at that time. Moreover, comments for declaration parts and scopes did hardly exist. The answer of the company on our report was that they are payed for a *working* software system and *not for documentation* and that commercial projects in general cannot spend much effort on source code documentation. A comparison with data from other systems, however, convinced the development team that their quantity of comments nevertheless lags behind industrial standards. Again, our analysis proved itself useful to identify weaknesses in terms of software quality and provide reference data to assess the deficiencies in contrast to other projects.

Assessing the commenting process

To understand source code and prevent bugs, it is important to keep comments up-to-date (Tan *et al.*, 2007). With our third experiment we can assess whether comments are kept up-to-date or at least adapted several revisions after the associated source code entity changed. That shows whether re-documentation is an integral part of the development process. For instance, we experienced that in ArgoUML, Eclipse Core, Eclipse PDE, and the Webframework re-documentation for declaration parts took place (see Table 6.4). The sooner the comments are adapted to source code changes the better we assess the commenting process of a system. But we also approve re-documentation because source code comments are added better late than never.

It is not necessary that every change induces a comment adaptation. In particular, different change types impact the consistency between comments and source code differently. Source code change types let us assess whether developers are aware of these different impacts. As the impact factor we use the *change significance level* that is assigned to each change type (Chapter 3). Concerning changes of scope comments we sum up the change significance levels of the change types applied inside the scope. The higher the significance level the higher the probability that the comment has to be adapted and the sooner this should take place.

Experiences with industrial partners. In the Webframework we found that a significant amount of declaration parts were re-documented. We know that the company

employed a person mainly for the re-documentation. This decision enhanced the quality of the commenting process.

The company of the other mentioned software system does not re-document declaration parts. The statement of the company was that as soon as the code works, it is not touched anymore—whether the API is commented or not does not matter.

Feedback during evolution

Beside the assessment of the quality of the commenting process of a software system, we further benefit from our investigations. We can provide feedback during evolution with a recommender that suggests when a developer might adapt the comments to source code changes. The change significance levels are then used to decide whether a suggestion is appropriate and to give a certain level of confidence. For instance, assume a developer makes several changes in an if-statement. When the sum of the change significance levels exceeds a specified threshold we can automatically suggest to adapt the comment of the if-statement (unless one exists).

6.3.2 Assessing our Mapping Approach

To map comments to source code, we have chosen a set of heuristics as described in Section 6.1.1. The heuristics are straight-forward, easy to understand, and reflect common practice, as confirmed by industrial partners. Nevertheless, we discuss issues of the mapping that might have influenced the results of our investigations.

Using heuristics to map comments to source code. The proximity between comments and source code is the most influencing factor for mapping a comment to a certain source code entity. In cases where the proximity heuristics are equal for both entities (succeeding and preceding the comment), we use a token based string similarity measure to check whether identifiers in the candidates appear in the comment. We did not validate whether using such a similarity measure reflects the semantical relation between the source code and the comment. However, as most mappings can be decided by using the proximity heuristics, we can accept this uncertainty factor.

Mapping comments to single source code entities. Our approach maps a comment to single source code entities. The limitation of this methodology is that not every comment describes a single source code entity. Developers also use comments to describe source code blocks, *e.g.*, sequences of statements. We partly cover this

practice by treating changes to scope comments but we miss sequences of simple statements. Consider the following illustrative example:

```
// Button to save using 'this' as selection listener
Button button = new Button(parent, SWT.NONE);
button.addSelectionListener(this);
button.setText("Save");
```

The comment describes a sequence of three statements. Our approach maps the comment to the variable declaration statement. A source code co-change for the comment only happens when the variable declaration statement changes together with the comment. There is no co-change, when, for instance, the selection listener is changed:

```
// Button to save using OpenFileChooser as selection listener
Button button = new Button(parent, SWT.NONE);
button.addSelectionListener(new OpenFileChooser());
button.setText("Save");
```

There are two possible solutions to overcome this situation. First, additional line delimiters that format the source code can split sequences of statements. Second, we may use the assumption that statements in-between two comments are described by the first comment. However, both possibilities are inappropriate to implement. Using delimiters or comments to split related statements depends on the coding conventions of a development team in general and on the practices of a single developer in particular. Extracting these conventions manually—even automatically—is not feasible. Moreover, there is no guarantee that such conventions or practices are applied consistently. This additional uncertainty factor would harm the validity of the results tremendously.

Incomplete mapping of comments to source code. Due to our approach of processing triples, we are currently not able to establish a proper mapping whenever successive comments are in different scopes. We always expect that a comment is among two source code entities; either on the class body or on the method body level. If not, then comments are related to comments, and comment changes to comment changes instead of source code changes.

This drawback results in comment changes that are due to comment changes. The change type *other* in Figure 6.5 (a) shows the percentages of such comment changes. In detail, these proportions are 15% for ArgoUML, 6% for Azureus, 3% for Eclipse Core, 15% for Eclipse JDT, 2% for Eclipse PDE, 4% for jEdit, 6% for JFreeChart, and 2% for Webframework. For each comment change the change of its associated source code entity is counted; changes between comments are therefore counted twice. Hence, the resulting error rate is between 1% and 8%, which we

consider acceptable for such an experiment. On the declaration level, the change type *other* does not have any impact (see Figure 6.5 (b)).

Not all comment changes are induced by source code changes. For all systems, over 50% of all comment changes where induced by changes of their associated source code entities. We cannot expect that all comment changes are induced by source code changes because of an external and an internal factor.

- *External factor*: Assume an interface without any API comment is added into the CVS repository and receives Revision 1.1. Its source code was not changed in Revision 1.2 but each method declaration was commented with an API comment. These inserts are not induced by any source code change because no changes are recorded for Revision 1.1. A similar scenario can happen for other source code entities and normal comments as well.
- *Internal factor*: Reconsider the `Button` example that explained the associated issue. In that example the comment change was not co-changed with a source code.

Analyzing whether the internal or the external factor has a higher impact on the number of comment and source code co-changes is subject of future work.

6.3.3 Threats to Validity

There are two major threats to the validity of this work.

Systems examined might not be representative. We examined eight software systems from different domains including one commercial system. It is still possible that we have chosen an unrepresentative set of systems for our study. However, the chosen open-source systems are well-known systems in the software evolution research community—especially ArgoUML, Azureus, Eclipse, and jEdit. Moreover, they have a rather long version history (3–7 years) to show a certain consistency in the commenting process. We found similarities as well as a certain diversity in the results of the three experiments. It is even appropriate to use three different components from the Eclipse software system. Eclipse is developed all over the world and is big enough to have a diversity in development. Consider the number of developers of the three components: Core has 47, JDT 55, and PDE 23 authors. The overlap of developers between them is as follows: Core and JDT have 19 common developers, Core and PDE have 10 common developers, and JDT and PDE have 7 common developers.

Our result indicate that the commenting process of the commercial system is comparable to the commenting process of the open-source systems.

All systems are written in Java. Extracting source code and comment changes on the AST level requires a complete programming language parser. As a result CHANGEDISTILLER currently supports the Java language. Systems in other object-oriented programming languages may be commented differently. However, we claim that the investigation of the commenting process of software systems is independent from the object-oriented programming language because common object-oriented languages provide similar language constructs for adding comments and commenting source code either depends on the development conventions or on the mood of developers.

6.4 Résumé

To gain a deeper understanding on how developers maintain source code documentation, we addressed three research questions in this chapter. In particular, we examined the question whether developers comment their code and to which extent they add comments or adapt them when they evolve the code.

We presented an approach to associate comment with source code entities to observe their co-evolution over multiple versions. A set of heuristics were used to decide whether a comment is associated to its preceding or its succeeding source code entity.

In summary the investigation of the research questions yielded in:

1. The growth factor of source code and comments are similar over time in all investigated software systems. But this does not directly mean that newly added code is well commented because half of the investigated systems have a commented source code proportion of less than 50 percent. It rather means that the ratio of comments to source code remains stable.

2. Whether a source code entity gets comment or not depends on its type. We even observed a partial order in the likeliness of whether a certain source code entity get commented.

3. Over 50 percent of comment changes are induced by source code changes. For six out of eight investigated systems over 90 percent of these co-changes are applied in the same revision.

The results have shown that our approach permits a quantitative assessment of the commenting process in a software system. We have successfully applied our toolset in projects with industrial partners to draw conclusions on different documentation-related facets of the quality of their software systems and their development processes. Furthermore, we argued that we can leverage the results to provide feedback during development to increase the awareness when to add comments or when to adapt comments because of source code changes.

In this chapter we have shown that the analysis of co-changes between comments and source code entities enables the reasoning about the commenting process of developers in a software system. We therefore accept Hypothesis H2a, and we regard the first part of Research Goal G3 as fulfilled.

Discovering Change Type Patterns 7

SOURCE code changes are rarely applied separately. In most cases a change induces other changes. For instance, a parameter renaming impacts all statements that access the parameter inside the method body. The statements have to be adapted to the preceding change. Kim *et al.* (2007a,b) extracted such refactorings and their corresponding changes in the method body to represent change patterns as rules. Other change pattern investigations were conducted to reveal error patterns (Kim *et al.*, 2006b; Livshits and Zimmermann, 2005) or aspect patterns (Breu and Zimmermann, 2006). Because of their interesting findings we explore whether change types appear frequently together, and whether they describe special development activities such as code cleanups, control flow paradigm shifts, or flow alterations. We present a semi-automated approach to discover change type patterns in the evolution of a software system using agglomerative hierarchical clustering.

We have performed experiments on one commercial and two open-source software systems to discover change type patterns. The results show that change patterns can be categorized into control flow change patterns, exception flow change patterns, and API change patterns. For instance, in the commercial system we discovered a remarkable shift from using *multiple exits* to *single exit* in methods. Moreover, our approach can discover that control flow changes are due to particular source code cleanup activities, that exception flow is used differently in different system parts, and that API convention changes are spread over many releases. For that, we have to distinguish between changes along the complete history and those that happen in certain periods. We categorize the found patterns and provide a catalogue of change type patterns.

In summary, this chapter makes the following contributions:

- It presents a semi-automated approach to discover patterns of change types.
- Based on the experimental study it presents a catalogue of change type patterns that describes which change types have to be applied and what rules must be fulfilled for a certain change pattern. This describes how such patterns can be automatically discovered and presented to the developer.
- It presents an experimental study with one commercial and two open-source software systems. We show that change type patterns can be separated into control flow, exception flow, and API change patterns.

The remainder of this chapter is structured as follows. We present our semi-automated approach to discover patterns of change types in Section 7.1. We apply the change type pattern extraction on three software systems in Section 7.2. From our findings, we present a catalogue of change type patterns in Section 7.3. We discuss the results and assess our approach in Section 7.4. We conclude this chapter with a reflection on the findings with respect to the hypotheses and research goals of this dissertation in Section 7.5.

7.1 Extraction of Change Type Patterns

In this section we describe the data that we extract and process to form change type patterns. We start by giving an overview of our approach (see Figure 7.1).

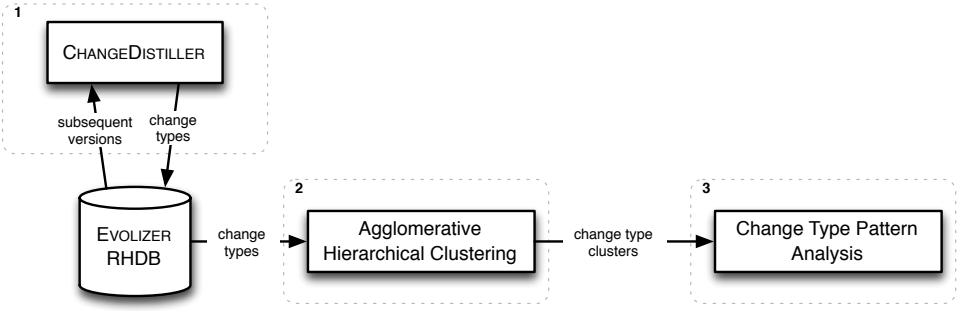


Figure 7.1: Overview on the change type pattern extraction process

1. We use ChangeDistiller (see Chapter 5), an implementation of our change distilling algorithm (see Chapter 4) to extract all source code changes from

the revisions stored in the EVOLIZER RHDB. The resulting change types are stored in our EVOLIZER RHDB.

2. For each method version, we fetch the change types and aggregate them in a matrix. This matrix is then used to build a cluster of change types using agglomerative hierarchical clustering.

3. We manually analyze the clusters to extract change type patterns. Finally, we express them as rules of change types.

7.1.1 Change Extraction and Change Types

Source code changes are extracted and classified by our change distilling algorithm (see Chapter 4). The classification is based on our taxonomy of source code changes which defines source code change types according to tree edit operations in the abstract syntax tree (AST) (see Chapter 3). In particular, our change distilling algorithm applies tree differencing pairwise on subsequent versions of ASTs of classes to extract the tree edit operations (see Chapter 4). We have implemented the algorithm in the Eclipse plugin CHANGEDISTILLER and it currently works for the Java programming language (see Chapter 5).

7.1.2 Clustering for Pattern Extraction

Our assumption is that a recurring coding activity is reflected by the same specific group of change types. Therefore our goal is (1) to identify those groups of change types that appear frequently together and (2) to describe the semantics of such groups by change type patterns. To achieve these goals we apply *cluster analysis*, a technique for identifying items within a data set that belong together. This section gives an overview of the cluster analysis technique we use. We then describe the approach to analyze change type clusters.

Overview of clustering technique

Assume a small program comprises four methods, $m_a, ..., m_d$. Each method was changed twice, so that each method has two versions, *e.g.*, m_a^1 m_a^2. Between the two subsequent versions of a method, m_a^1 and m_a^2, we extract the change types with CHANGEDISTILLER and attach the change types to m_a^2, *i.e.*, the change types that transform m_a^1 to m_a^2 are attached to m_a^2. We build a matrix where change types denote the rows and method versions the columns. A matrix entry is then the number of occurrences of a change type in a method version. The result of this operation for method versions $m_a^2, ..., m_d^2$ is depicted in Table 7.1. For instance, method version m_b^2 had one *return type delete* and two *statement deletes*.

	m_a^2	m_b^2	m_c^2	m_d^2
Parameter delete	2	0	1	0
Return type delete	0	1	0	0
Statement delete	1	2	0	1
Statement insert	0	0	3	1
Statement update	4	0	1	2

Table 7.1: Example matrix used for clustering

Distance measures. To perform cluster analysis on change type data, we need a way to describe whether certain change types belong together, *i.e.*, whether they occur together over time. Clustering normally uses *association coefficients* or *distance measures* to express proximity between items of a data set (Jain *et al.*, 1999; Maqbool and Bari, 2007).

We treat each row of the matrix as a spatial vector—each change type occupies a position in an n-dimensional space. The dimension is defined by the number of method versions. Two well-known measures to compare spatial entities are *Euclidean* and *Cosine* distance measures. We use the Cosine measure to calculate the distances between the change type vectors. The Cosine distance is defined as follows:

$$d_c(x_i, x_j) = \left(1 - \frac{x_i \cdot x_j}{\|x_i\|\|x_j\|}\right)$$

where x_i, x_j are the row vectors of row i and j, and $\|x\|$ is the *Euclidean norm* of vector x. The Euclidean distance is defined as follows:

$$d_e(x_i, x_j) = \|x_i - x_j\| = \sqrt{\sum_{i=1}^{n}(x_{i,k} - x_{j,k})^2}$$

where n is the dimension of the vectors and $x_{i,k}$ is the kth dimension value of x_i.

We prefer the Cosine distance over the Euclidean distance because it relies only on the direction of the vectors. Consider the vectors v_1 and v_2 in Figure 7.2.

The angle between them is $\varphi = 15.3°$; the distances are $d_c(v_1, v_2) = 0.04$ and $d_e(v_1, v_2) = 12.2$. Since the Cosine distance is a function of the angle between the vectors, *i.e.*, the length of the vectors does not influence the Cosine distance measure, it is smaller than the Euclidean distance. This is preferable for calculating

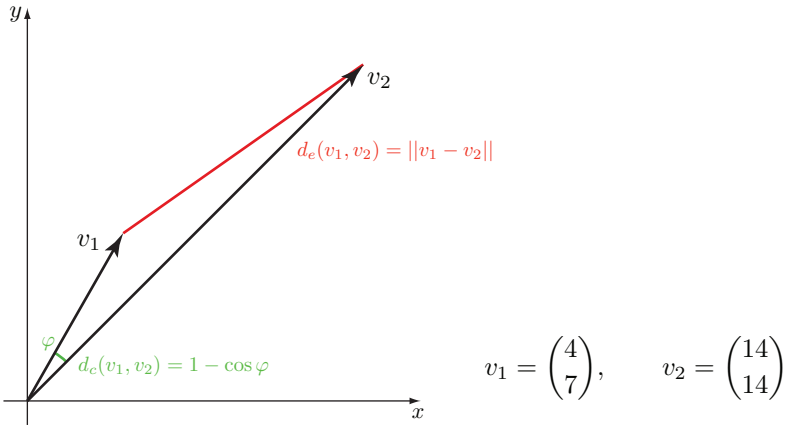


Figure 7.2: Example of the difference between the Euclidean distance measure (red) and the Cosine distance measure (green, as a function of the angle between the vectors)

the distances between change type vectors because it matters *that* certain change types occur together and not *to what extent* in the first place. But taking binary vectors, *i.e.*, 0 if the change type does not occur and 1 otherwise, is also not preferable, because the direction of the original vector disappears.

The distances between any two change types is put in a *dissimilarity* (or *distance*) matrix. The dissimilarity matrix for Table 7.1 is shown in Table 7.2.

	Param. del.	Ret. type del.	Stmt. del.	Stmt. ins.	Stmt. upd.
Parameter del.	0	1.0	0.635	0.576	0.122
Return type del.		0	0.184	1.0	1.0
Statement del.			0	0.871	0.466
Statement ins.				0	0.655
Statement upd.					0

Table 7.2: Dissimilarity matrix of Table 7.1

Agglomerative hierarchical clustering. Agglomerative hierarchical clustering algorithms start with each item in a single cluster and then iteratively links clusters together to a new cluster until only one cluster remains. The linking is done with a *linkage* function. For the clustering of change types we use the *average linkage* between two clusters c_a and c_b. Average linkage uses the average distance between

all pairs of elements in cluster c_a and cluster c_b:

$$link_{al}(c_a, c_b) = \frac{1}{|c_a||c_b|} \sum_{i=1}^{|c_a|} \sum_{j=1}^{|c_b|} d_c(c_{a,i}, c_{b,j})$$

where $|c_a|$ is the number of elements in the cluster c_a, and $c_{a,i}$ is the ith element in the cluster c_a. The result of the hierarchical clustering can be visualized by a *dendrogram*. The dendrogram of our example is depicted in Figure 7.3.

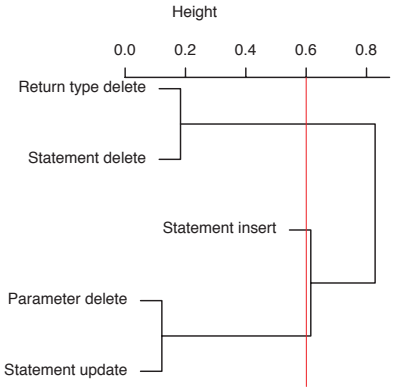


Figure 7.3: Resulting dendrogram for the clustering of Table 7.2. The red line indicates the cutoff

Having a dendrogram as shown in Figure 7.3, we can specify a *cutoff* value, meaning that we cut the dendrogram by a horizontal line at the height of the cutoff value. The dendrogram is then read from left to the cutoff line and the yielding complete branches are then the resulting subclusters, *i.e.*, in our case change type clusters. Assume we specify the cutoff value to 0.6 in Figure 7.3, the resulting change type clusters are: (1) {return type delete, statement delete}, (2) {parameter delete, statement update}, and (3) {statement insert}. The third cluster is comprised of only one change type because the connection from statement insert to the {parameter delete, statement update} cluster is above the 0.6 cutoff.

Explanation of the dendrogram. The axis labelled "Height" in the dendrogram in Figure 7.3 shows the relative distance the clusters have from each other. The change types *parameter delete* and *statement update* have a distance of 0.122, as listed in Table 7.2. They form a cluster, c_1. The change types *return type delete* and *statement delete* have distance 0.184 and also form a cluster, c_2. Using the average link-

age, it is calculated whether *statement insert*, si, is attached to c_1 or c_2:

$$link_{al}(c_1, si) = \frac{1}{2 \cdot 1}(0.576 + 0.655) = 0.616$$

$$link_{al}(c_2, si) = \frac{1}{2 \cdot 1}(1.0 + 0.871) = 0.936$$

Because of $link_{al}(c_1, si) < link_{al}(c_2, si)$ the statement insert is clustered to c_1 at the height of 0.616, resulting in cluster c_3. The final cluster, c_4, is the combination of c_2 and c_3 at the height

$$link_{al}(c_2, c_3) = \frac{1}{2 \cdot 3}(1.0 + 1.0 + 1.0 + 0.871 + 0.635 + 0.466) = 0.829$$

For a more detailed discussion of data clustering in general and hierarchical clustering for software engineering in particular we refer to (Jain *et al.*, 1999) and (Maqbool and Bari, 2007).

7.1.3 Analysis of Change Type Patterns

For the analysis of change type patterns we perform several clustering passes. The first pass takes the change types during the entire history of a software system into account. We take all method versions that are extracted from CHANGEDISTILLER, create a matrix analogously to Table 7.1 described in Section 7.1.2, and perform the clustering on it. For the second and all further passes we divide the change history of a software system into yearly quarters and build the matrix of method versions for each of these quarters. For the remainder of this chapter we distinguish these two kinds of cluster passes as *full cluster* and *quarter cluster*.

After building the full and the quarter clusters, we make a two step analysis. First, we analyze the change type patterns of the full cluster. We define the patterns that we reveal from the full cluster as *global* change type patterns as they can be found when analyzing the entire history. Second, we analyze and compare the change type patterns of the quarter clusters among each other and with those of the full cluster. As the data extracted for calculating the full cluster contains a large amount of method versions, deviations of global change type patterns applied during the history of a software system disappear. Because of that we aim at finding *local* change type patterns within the quarter clusters—patterns that deviate from the global patterns.

System	# source revisions	# changes	LOC first	LOC last
jEdit	6,754	88,932	80,726	133,895
JFreeChart	4,675	23,678	151,040	250,180
Webframework	19,501	116,994	43,452	124,796
Total	30,930	229,604	275,218	508,871

Table 7.3: Analyzed software systems. [# source revisions] indicates the total number of revisions of Java files. [# changes] indicates the number of changes type occurrences that were applied during the period. [LOC first] and [LOC last] indicate the lines of code for the first and the last release of the component in the period

We do not list predefined change type patterns in this section. Instead, we collect and describe them in Section 7.2. We use the findings to present a catalogue of change type patterns in Section 7.3. In the catalogue we categorize the change type pattern and express them in rules of change types.

7.2 Experimental Results

In this section we describe the results of applying our change type pattern extraction approach to three case studies. In Section 7.2.1 we present the experimental setup; in Section 7.2.2 we evaluate the results and describe our findings. We conclude this section with a summary of our findings in Section 7.2.3.

7.2.1 Experimental Setup

We have chosen two open-source and one commercial system for our experiment:

1. jEdit (text editor; observation period: Sep 01 – Jun 06)

2. JFreeChart (Java chart library; observation period: Oct 01 – Jul 07)

3. Webframework (a commercial framework for web applications; observation period: Jul 04 – Sep 07)

All projects are written in Java and are version-controlled using CVS. jEdit and JFreeChart already moved to Subversion. For jEdit we received an older repository directly from the developers and for JFreeChart, the CVS repository is still available on sourceforge.net. Table 7.3 summarizes these software systems.

7.2.2 Discovered Change Type Patterns

For all three case studies, we created the full change type clusters, *i.e.*, for the entire history, and the quarter change type clusters, *i.e.*, for every quarter of a year. First, the full change type clusters are discussed and the global change type patterns are presented. We then present our findings of the local change type patterns. We found more interesting change type patterns in the Webframework than in jEdit and JFreeChart.

Global patterns in the Webframework

The full cluster of the Webframework is depicted in Figure 7.4 as an exemplar of change type clusters. All other clusters that we discuss in this chapter are shown in Appendix E. We set the cutoff value to 0.6.[1] We build the change type patterns, as described in Section 7.1.2: We read the dendrogram topdown and from left to right until we reach the cutoff line. We do not list all possible change type patterns that can be found in the dendrogram. Instead we shortly describe those that are interesting.

Constructor invocation changes. Super constructors are invoked when the super class provides constructors that are not overridden by the class. This change type pattern is applied, when the super constructor is overridden in the class. Then, the super constructor invocation has to be replaced by a constructor invocation. Since the two change types are clustered at approximately 0.5 the pattern either appears seldom or the change types are not applied together every time.

Return type based method renaming. In the Webframework, developers tend to change the method name when the return type of the method changes. This is indicated by the change type cluster {return type change, method renaming}. This is reasonable because the return type carries important semantics. The two change types are clustered closely to the cutoff line because we can expect that not every method renaming induces a return type change.

[1] We discuss how to choose an appropriate cutoff value in Section 7.4.2

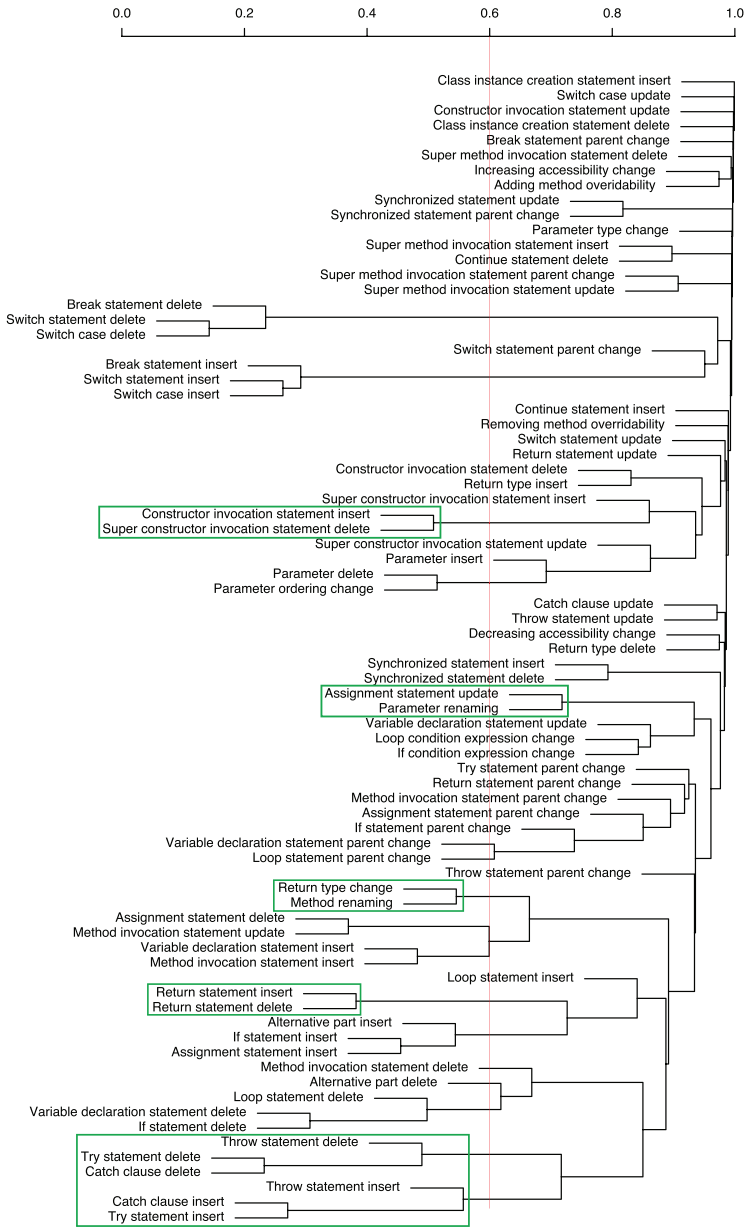


Figure 7.4: Full change type cluster of Webframework. The red line indicates the cutoff. Change type patterns discussed in the text are highlighted with green rectangles

Introducing prefixed parameter names. Although the change types {parameter renaming, update assignment} are clustered above the cutoff value, they are interesting to discuss. The change type update assignment occurs often and in various conjunction with other change types. It is, therefore, remarkable that it is clustered with the change type parameter renaming—a rather infrequent change type.

To find the reason for this change type pattern, we compared the full cluster with the quarter clusters. In six out of 13 quarter clusters of the Webframework update assignment and parameter renaming are clustered, but only twice below the cutoff of 0.6—in the first quarter of 2005 and in the third quarter of 2006. We inspected the occurrences of the change types manually and found that in 285 out of 493 parameter renamings the parameter name got an undefined article as a prefix; for instance `root` became `aRoot`. This occurred mostly in setters and constructors where assignments initialize field values.

Developers can make such changes with the support of an IDE such as Eclipse. However, the changes should not occur when naming conventions for parameters exist at the beginning of the software project and are strictly applied throughout its life-cycle. Since the changes did not happen within a certain period but are rather spread over the whole observation period, we can exclude that there was a shift in the naming conventions.

Introducing single exit. It strikes us that return statement insert and delete are linked around 0.4. A possible explanation that they are frequently applied together is the known issue in our change distilling algorithm (see Chapter 4 for a discussion): When the string similarity between two statements is below a certain threshold, the algorithm stores a statement insert and delete instead of a statement update. This can also happen for return statements and when it happens frequently the change types return statement insert and delete have to be clustered together. But as they are further clustered with if-statement, else-part, and assignment insert, other reasons may be possible.

After an inspection of the method versions in which the developers of the Webframework applied these change types, we found that there was a shift from the *multiple exit* to the *single exit* principle. Consider the following example: The method

```
public boolean hasAnyNextSteps() {
    if (getCurrentStep() instanceof StepSelectionStep) {
        return true;
    }
    return configuration().hasAnyNextSteps(getCurrentStep());
}
```

was restructured to

```
public boolean hasAnyNextSteps() {
   boolean hasAnyNextSteps = false;
   if (getCurrentStep() instanceof StepSelectionStep) {
      hasAnyNextSteps = true;
   } else {
      hasAnyNextSteps = configuration().hasAnyNextSteps(
         getCurrentStep());
   }
   return hasAnyNextSteps;
}
```

By examining the quarter clusters, we found that the developers heavily introduced this single exit principle in the third and fourth quarter of 2005. We were then wondering whether they kept this principle during 2006 and 2007. The answer is twofold. First, they kept introducing the single exit principle. Second, although they were already aware of the validity of the principle in the Webframework, they also made changes violating the principle. A further investigation showed that they sometimes corrected these violating changes within 100 days, but hardly for all of those affected methods.

Change existing exception handling. Inserting and deleting try statements mostly effect inserting and deleting catch clauses as well, but they do not always have to be applied together. For instance, an additional catch clause can be added to a try statement after the try statement was inserted. Therefore, try statement insert/delete are well clustered (*i.e.*, below 0.4) with catch clause insert/delete, but not perfectly (*i.e.*, at 0.0). The change type delete throw statement is attached on the cluster {delete try statement, delete catch clause}, but not closely as illustrated by the difference in height. Analogously, throw statement insert is clustered to insert of try-catch-statements. The reasons for that are twofold:

1. Exceptions are not only handled with re-throwing them. There must be cases in which the exception is, for instance, caught and a stack trace is logged.

2. During the development of the framework, the developers decided to make a shift from or towards re-throwing exceptions. Either they introduced Webframework specific exceptions to collect them at a certain point or they introduced logging to handle exceptions appropriately.

Both have the effect that no try-catch-statement is changed along with a corresponding throw statement over the history of the Webframework.

After an inspection of the changes in the source code we found out that there was not any shift in the exception handling, the developers rather used two different mechanisms to handle exceptions: (1) They throw a new Webframework

defined exception or (2) they log the exception. These are common practices to handle exceptions. However, there are methods that handle exceptions without rethrowing them but declare to throw exceptions. It seems that exceptions still might pass through these methods without any handling. This makes the source code hard to understand because it is not clear—and also not documented—in which situation an exception is caught and handled, and when it is passed through.

Local patterns in the Webframework

In this section we describe the change type patterns in the quarter clusters that deviate from the patterns in the full cluster.

Swap control flow order. In the third and fourth quarter cluster of 2005, the statement parent changes are grouped together with control structure condition expression change (see Appendix E). Compared to the full cluster in Figure 7.4, the parent changes are indeed clustered but on a high level in the dendrogram. In addition, the control structure condition expression change is far apart from them in the dendrogram.

An inspection of the changes in the source code revealed that the parent change pattern mostly denotes swapping the then and else-part of an if-statement. That leads to the following changes: The if-condition must be negated (control structure condition expression change), the statements in the then-part are moved to the else-part (statement parent change) and vice versa. A reason for this change pattern is the convention that the default control flow goes via the then-part. For instance the method

```
public void print() {
    DocumentFile documentFile = getDocumentFile();
    if (selectedPrinter != null) {
        OutputManager.getInstance().print(documentFile,
            selectedPrinter);
    } else {
        OutputManager.getInstance().print(documentFile);
    }
}
```

was restructured to

```
public void print() {
    DocumentFile documentFile = getDocumentFile();
    if (selectedPrinter == null) {
        OutputManager.getInstance().print(documentFile);
    } else {
```

```
            OutputManager.getInstance().print(documentFile,
                selectedPrinter);
        }
    }
```

Although such control flow order changes appear in the second half year of 2005 concentratively, they can be sporadically found over the history of the Web-framework.

Merging control flow. In the first and second quarter of 2007, the developers made another kind of control flow change (see Appendix E): The change type cluster {if-statement parent change, control structure condition expression change} appears in the first and second quarter cluster of 2007. We inspected the changes in the source code and found out that a certain amount of nested if-statements were merged. For instance, the method

```
public Model getObject(Session aSession) {
    if (obj == null) {
        if (getExClassId() != null && getExOid() != null) {
            ...
        }
    }
    return (Model) obj;
}
```

was restructured to

```
public Model getObject(Session aSession) {
    if (obj == null && getExClassId() != null &&
        getExOid() != null)
    {
        ...
    }
    return (Model) obj;
}
```

The inner if-statement was moved out of the outer if-statement (if-statement parent change), the conditions of the two if-statements are merged (control structure condition expression change), and the outer if-statement was deleted (if-statement delete). Parent changes of further statements inside the inner if-statement can also have been applied.

Remove superfluous parameter. In the third quarter cluster of 2005, the change types parameter delete and return statement update are grouped (see Appendix E).

A considerable amount of XML handling functionality is provided via delegators in the Webframework. During a short period in the third quarter, the developers removed the session parameter from various methods and adapted the return statements accordingly. For instance, the method

```
public XmlTree getRequestTree(
   Properties requestProps,
   Session aSession)
{
   return getRequestTree(requestProps, aSession,
      new HashMap());
}
```

was changed to

```
public XmlTree getRequestTree(
   Properties requestProps)
{
   return getRequestTree(requestProps, new HashMap());
}
```

Change type patterns in jEdit and JFreeChart

Additionally, we only found two further change type patterns in the JFreeChart and jEdit. Both dendrograms are shown in Appendix E.

First, in JFreeChart, the developers use exception flow to check method preconditions. Basically, they check the parameter values for certain conditions. As a result, if-statement and throw statement inserts are grouped in the full cluster. For example, in the method

```
public void setCategory(Comparable category) {
   this.category = category;
}
```

they inserted a parameter check:

```
public void setCategory(Comparable category) {
   if (category == null) {
      throw new IllegalArgumentException(
         "Null 'category' argument.");
   }
   this.category = category;
}
```

Second, in jEdit we found the change type pattern {if condition expression change, variable declaration update} during the first quarter of 2005. This pattern did not appear in the full cluster. The developers removed direct field accesses with getters.

7.2.3 Summary of Experiments

Clustering of change types reveals patterns of: (1) Changes in the control flow, (2) changes in the exception flow, and (3) changes in the API.

Furthermore, we revealed that (1) control flow changes are due to particular source code cleanup activities, (2) exception flow is used differently in system parts, and (3) API convention changes that are spread over many releases.

The results showed also that special change type patterns are rare and that the found patterns are rather due to corrections of coding guideline violations than other feature driven changes.

7.3 A Catalogue of Change Type Patterns

Based on the findings in the experiments we created a catalogue of change patterns. The catalogue consists of three categories: (1) Control flow change patterns, (2) exception flow change patterns, and (3) API change patterns. We describe which change types have to be applied and what rules must be fulfilled that a certain change pattern occurs. It is important to know, that finding change types fulfilling the rules does not necessarily mean that the expected change pattern was applied. The rules express the indication for a certain pattern. Where appropriate we also include class body change types, such as attribute or method insert.

Change type pattern	*Involved change types*	*Rule*
Control flow change type patterns		
Constructor invocation change	super constructor invocation delete, constructor invocation insert	-
Introducing single exit	return statement insert, return statement delete, variable declaration insert	no return type insert, # return statement insert $<$ # return statement delete

Change type pattern	*Involved change types*	*Rule*
Swap control flow order	control structure condition expression change, parent change of any statement	statements of else-part are moved to then-part or vice versa
Merge control flow	if-statement parent change, control structure condition expression change, if-statement delete, parent change of any statement	if and other statements are moved into an if-statement, new condition expression must contain the condition of deleted if-statement
Exception flow change type patterns		
Change existing exception handling	throw statement insert, delete of any statement	throw statement inserted in the same catch clause as the statements are deleted
Introducing exception precondition checking	if-statement insert, throw statement insert	if-statement inserted as first statement in method, throw inserted in if-statement
API change type patterns		
Return type based method renaming	return type change, method renaming	-
Introducing prefixed parameter names	parameter renaming	old parameter name is prefixed and not further changed
Remove superfluous parameter	parameter delete, update of any statement	old statement contains parameter
Encapsulate field	update of any statement, method insert	new method name is `get<field_name>`, new statement contains call to method

7.4 Discussion

Our change type clustering approach revealed interesting change type patterns. In particular we did not expect to find semantic-preserving changes in control flows. On the other hand, although we investigated industrial and open source software systems, the outcome is rather moderate. In this section, we discuss how we can benefit from extracting change type patterns and whether the technique we used to extract them is appropriate or what alternatives do exist.

7.4.1 Benefits from Change Type Patterns

We can leverage our patterns for several scenarios:

Catalogue of change type patterns. Our catalogue provides meaning to particular changes during the evolution of a software system. Since software feature development is mixed with source code cleanups, we are able to look for specific patterns and isolate them in certain time periods.

Consistency of changes. Discovering change type patterns enables a consistency analysis of the source code changes, especially when paradigm shifts take place. For example, the introduction of the *single exit principle* in the Webframework started in a specific period and should have been implemented consistently in all parts of the architecture. With our approach we are able to discover this point in time and in checking this we were able to find violations of this principle. We can, therefore, leverage our approach to make developers aware of inconsistent changes.

Feedback during evolution. As most change type patterns can be seen as code cleanup changes, one might argue that they are not exciting and obviously happen during software development. But, we can also learn from these patterns: Either coding guidelines are adapted frequently or they are not followed strictly. Revealing inconsistencies in applying coding guidelines is an important part of software quality assurance. We can provide feedback during evolution with a recommender that can be configured either by users or by learning from the occurred change type patterns. Moreover, a recommender makes programmers, who are new to a software project, aware of certain guidelines and support their fast adoption and correct usage.

Semi-automated analysis. For the population of a catalogue, our approach is automated up to the interpretation of the change type dendrograms. By specifying a

cutoff value we are even able to fully automate the extraction of groups of change types that appear frequently. Using the specified change type patterns, we can search for specific occurrences automatically and provide checks and feedback for the above scenarios.

7.4.2 Assessing our Approach

A reason for the moderate outcome of our experiments—moderate in the sense that we expected to find more interesting change type patterns—might be that agglomerative hierarchical clustering of occurring change types is suboptimal. So we discuss advantages and drawbacks of this technique to assess its accuracy.

Advantages. Agglomerative hierarchical clustering relies on a distance measure between two elements. It groups elements that have a short distances, *i.e.*, are close to each other, in the first place. Our distance measure reflects primarily whether change types appear together and secondarily to what extent. As we want to group change types that appear frequently together, this distance measure is appropriate for our concerns. Assume two change types are perfectly clustered, *i.e.*, have a distance of zero. This happens when the change types appear in exactly the same methods and to the same extent. Thus the hierarchical clustering groups the change types correctly. Moreover, the change type patterns that we can extract from dendrograms, such as depicted in Figure 7.4, make sense: Consider the change type pattern {switch statement insert, switch case insert, break statement insert} in the upper half of the dendrogram. It is perfectly reasonable that these change types are grouped in the cluster because a switch statement consists of switch cases and break statements. It is also valid that the three change types are not grouped on the same height. A developer can add a new switch case or break statement to an existing switch statement. The advantage of agglomerative hierarchical clustering is that it produces occurring and correct change type patterns.

Drawbacks. Hierarchical clustering has one major drawback: It provides exactly one result, *i.e.*, the cluster in which linked change types have the shortest distance. These linked change types appear mostly together but not always; there are other possible combinations between them. Because of that we may miss interesting or important change type patterns. This effect is reinforced when developers mix feature development and cleanup changes between two commits. Since feature development involves the use of arbitrary change types, cleanup patterns are blurred by this mixture. This is the main reason why we only found cleanup patterns in the Webframework and not in the other two software systems. Developers in the Webframework committed single cleanup changes to the repository and even added a corresponding commit message.

An alternative to hierarchical clustering is *fuzzy clustering* (Jain *et al.*, 1999) that provides multiple clusters for the same change type. The probabilities of their occurrence is then attached to each of the clusters. That means, one change type can be in more than one pattern. For instance, reconsider the example we used in Section 7.1.2. According to Table 7.2 the distance between statement insert and statement update is smaller than between statement insert and parameter delete. By using fuzzy clustering, statement insert would not be in the same cluster as statement update and parameter delete. Instead, statement update would be in two clusters: {statement insert, statement update} and {parameter delete, statement update}. Applying fuzzy clustering is subject to future work.

Cutoff value. Choosing the appropriate cutoff value is not always obvious. If we choose the cutoff value to low, we either miss import patterns higher in the dendrogram or we have to deal with several small patterns. If we, on the other hand, choose the cutoff value to high, we obtain only a few large patterns. Consider the dendrogram in Figure 7.4. The parent change branch (*i.e.*, subcluster) on the right hand side of the dendrogram or the change type pattern {assignment update, parameter renaming} revealed interesting patterns but are above the cutoff line. But, using a bigger cutoff value leads to cluster in which we miss interesting subclusters. For instance, we then miss the change type pattern {method renaming, return type change}. To conclude, a cutoff value should guide our pattern search but not specify a single solution.

7.4.3 Alternative Pattern Extraction Techniques

Because of the major drawback of the hierarchical clustering we considered different change type pattern extraction techniques. We tested concept analysis (CA) and locality-sensitive hashing (LSH). We briefly describe their mechanisms, how we applied them to find change type patterns, and discuss the issues we encountered.

Concept analysis

We use the explanation of Siff and Reps (1999) to skim the mechanism of CA. According to Wille (1981), concept analysis provides a way to identify groups of objects that have common attributes. A concept is then defined as a pair of sets: The set of objects and the set of attributes. This means for our concern that *groups of method versions have common change types* and that the pair of sets is: The set of method versions and the set of change types. The cardinality of the set of method versions describes the frequency of a set of change types.

CA expects an adjacency matrix. A matrix entry is 1 if a change type (attribute) was applied to a method version (object) and 0 otherwise. To calculate the concepts

we used the transposed occurrence matrix (*e.g.*, see Table 7.1) and replaced every entry > 0 with 1.

After calculating the concepts for the Webframework, we decided to not use CA for our concern. The main reason is that the diversity of possible combinations of the used change types leads to thousands of concepts—too many to make an appropriate analysis.

Locality-sensitive hashing

Another approach to find similarities between change type vectors (rows in Table 7.1) is using locality-sensitive hashing (LSH) (Datar *et al.*, 2004; Gionis *et al.*, 1999). The LSH algorithm hashes the vectors and stores them in *buckets*. For every hash key a new bucket is generated. The core idea of the algorithm is to use hash functions with certain properties. The properties have to ensure that the probability of collision is much higher for vectors that are close to each other than for those that are far apart. We refer to (Jakob, 2007) for a more detailed discussion on discovering change type patterns using LSH.

The change type patterns that LSH revealed were hard to interpret because LSH has two drawbacks for our concerns. First, the outcome of the LSH is not deterministic. As the algorithm operates with probabilities, we obtain different results for each run on the same set of data. Intersections of different results are not reliable as well. Second, similar to CA, the diversity of possible combinations of change types leads to too many patterns.

7.4.4 Threats to Validity

There are two major threats to the validity of this work.

Systems examined might not be representative. We examined one commercial and two open-source software systems. It is, therefore, possible that we have chosen an unrepresentative set of systems for our study. Since the results of only one system of our study revealed significant change type patterns, this threat to validity is especially important. However, we have shown that change type patterns exist, that we can extract them, and that we revealed differences in commercial and open-source systems, we at least have a meaningful diversity in the data set.

As the outcome of the jEdit and JFreeChart case studies is moderate, we will perform further case studies to extend the catalogue of change type patterns as well as show their general validity.

All systems are written in Java. Extracting source code changes on the AST level requires a complete programming language parser. As a result CHANGEDISTILLER

currently supports the Java language. Systems in other programming language may have different change type patterns. However, we claim that the investigation of the change type patterns is independent from the programming language because on the statement level, programming language of the same paradigm are similar.

7.5 Résumé

We introduced an approach to explore whether change types appear frequently together, and whether they describe special development activities. The idea was to use agglomerative hierarchical clustering of change types to discover such patterns of change types.

We performed experiments on one commercial and two open-source software systems. The results showed that change patterns can be categorized into control flow change patterns, exception flow change patterns, and API change patterns. Furthermore, our approach can discover that certain control flow changes are due to particular source code cleanup activities, that exception flow is used differently in different system parts, and that API convention changes are spread over many releases. For that, we had to distinguish between changes over the entire history and such that happen in certain time periods (*i.e.*, yearly quarters). We categorized the found patterns and provided a catalogue of change type patterns. The catalogue describes which change types have to be applied and what rules must be fulfilled for a certain pattern. That also describes how such patterns can be automatically discovered and presented to the developer.

In this chapter we have shown that the analysis of change type patterns allows us to categorize change and development activities during the course of the evolution of a software system. But we have also realized that our approach is not robust enough against a strong mixture of development activities. We therefore accept Hypothesis H2b only partially, and we regard the second part of the Research Goal G3 as partially fulfilled.

Change-Affected Method Invocations

8

Brook's (1995) *essential complexities* in software engineering inhibit the development of bug-free software. Bugs are continuously fixed, but because fixing bugs induces software changes, each fix has a substantial chance to introduce a new bug. Finding and fixing bugs before the software is delivered to customers is of great value because fixing bugs after delivery is ten times as expensive as fixing it before (Davis, 1995). As a result, approaches and techniques that find bugs automatically and as soon as possible are crucial.

In the area of software evolution analysis, approaches exist that find bugs by mining error patterns in source code. For instance, Livshits and Zimmermann (2005) applied mining to software repositories to find errors in method usage patterns. Kim *et al.* (2006b) used their *BugMem* approach to find general error patterns by investigating changes that fixed bugs. The major advantage of such approaches is that they are vertical. That means, they find project-specific error patterns. Horizontal approaches, such as *FindBugs* (Hovemeyer and Pugh, 2004), find error patterns across all projects but are limited to predefined patterns.

Although *BugMem* can find a variety of error patterns, it fails at identifying move changes that fixed a bug. That means, moving a method invocation into the then or the else-part of an if-statement cannot be identified as a bug fix. As our change distilling algorithm can extract such *context changes* of method invocations we can extract corresponding bug fix changes and overcome this limitation.

We observed that in Eclipse a significant number of bugs are fixed by context and update changes of method invocations (see Section 8.1). In addition, these changes also happen even if they did not fix bugs, *i.e.*, they are also applied during preventive, perfective, or adaptive maintenance. To better understand the nature

of those changes we address the following research questions in this chapter:

1. Do methods exist whose invocations are significantly more affected by context changes and update changes than other methods? The knowledge of such methods is important to raise the awareness for which method invocations developers should consider a context or an update change.

2. Can we reveal change patterns among method invocations that are affected by context and update changes? We assume that patterns of context and update changes can be found for invocations of particular methods. In the long-term we can then leverage such patterns to support the use of method by suggesting context and update changes. With these suggestions we aim at reducing the number of bugs related to those invocations.

To answer these questions we developed an approach to rank methods whose invocations are affected by context and update changes. The rank reflects how often such changes were applied to invocations of a particular method and whether these changes were bug fixes. In addition, we extract patterns of such changes to discuss whether or not change suggestions could be reasonable.

We have performed experiments on three core components of Eclipse: Core, JDT, and PDE. We apply the ranking and extracted invocation change patterns. The results show:

1. String handling, output generation, and data structure methods of the Java SE Development Kit (JDK) are among the highest ranked methods in all three components.

2. We identified several patterns of method invocation changes. For instance, for invocations of the `List.add(..)` method, conditions that check whether the argument of the invocation is **`null`** or already in the list are frequently added when fixing bugs. The identified update change patterns most frequently adapt the arguments of the invocations.

In summary, this chapter makes the following contributions:

- It presents an approach to rank methods whose invocations are affected by context and update changes. The rank is calculated from information we obtain from the change history of a software system.

- It describes a technique to extract invocation change patterns of particular methods.

- It presents a detailed experimental study with three components of Eclipse. The Eclipse components comprise a change history of 6.5 years with more than a million source code changes. For Eclipse, we show that invocations of data structure methods of JDK are most affected by context and update changes and that invocations of methods exhibit certain change patterns.

- It discusses whether the ranking and the extracted patterns are appropriate for change suggestions to support developers.

In the remainder of this chapter we motivate our approach by presenting observations from the change histories of the Eclipse components in Section 8.1. We then describe the data extraction and preparation to rank the methods and present the change pattern extraction in Section 8.2. Section 8.3 shows how we applied the ranking and change patterns extraction on three core components of Eclipse. We discuss the results and assess our approach in Section 8.4. We conclude this chapter with a reflection on the findings with respect to the hypotheses and research goals of this dissertation in Section 8.5.

8.1 Motivation for Selected Change Types

The selection of the two change types for our approach hinges on the following observations. We made them while investigating source code changes that fixed bugs in the Eclipse project.

OBSERVATION 8.1 *25 percent of bugs are fixed with fewer than four source code changes.*

In Eclipse, over 50 percent of bugs only affect one Java file; 25 percent are even fixed with fewer than four source code changes (instances of change types). For instance, they were fixed with two *statement inserts* and one *control structure condition expression change*. This observation is supported by the finding that small changes, *i.e.*, single line changes, often represent bug fixes (Livshits and Zimmermann, 2005; Purushothaman and Perry, 2005). While extracting the source code changes that fixed bugs and analyzing their change types we made the next observation.

OBSERVATION 8.2 *Two change types are responsible for fixing a significant amount of bugs.*

Our change distilling algorithm extracts over 120 different change types (see Chapter 4).[1] Two of them are responsible for fixing 15, and 33 percent respectively, of all bug fixes, (1) moving a single method invocation inside the then or the else-part of an if-statement and (2) updating a method invocation, *e.g.*, changing an argument. By looking at the detailed change type information we found commonalities between the changes and the involved method invocations. The following two observations provide the basis for the hypothesis that we can learn from change histories to make change suggestions that support the use of methods.

[1] When distinguishing all possible statement kinds. For instance, the change type *statement update* can be further split into *method invocation statement update*, *assignment statement update*, etc.

OBSERVATION 8.3 *Invocations of a method tend to be changed similarly.*

Consider, for example, the method `List.add(..)` from the JDK. This method adds an object of any type to a list. Assume a set of invocations of this method conform to `<qualifier>.add(<argument>)`. This set of invocations is similarly updated if, for instance, the argument is replaced by a getter: `<qualifier>.add(anObject)` is updated to `<qualifier>.add(getAnObject())`.

OBSERVATION 8.4 *The contexts of invocations of a method tend to be changed similarly.*

The contexts of a set of `List.add(..)` invocations change similarly if these invocations are moved into the then or the else-part of an if-statement that has a similar condition. Such a condition can be `!<qualifier>.contains(<argument>)` or `<argument> instanceof <Type>`.

Based on the observation described above, we rank those methods whose invocations are affected by context and update changes. We also extract patterns among method invocation changes. In the long-term, we aim at providing feedback to developers with the data described in this chapter and reduce the number of bugs consequently.

8.2 Method Ranking and Pattern Extraction

To answer the two research questions we extract and collect source code change data. In this section we describe the processing of this data to rank methods and to extract change patterns of their invocations. We start by giving an overview of our approach (see Figure 8.1).

1. We process commit messages to find bug numbers and fetch corresponding bug reports from Bugzilla. We store the corresponding bug models including links to revisions in the EVOLIZER RHDB.

2. We use CHANGEDISTILLER (see Chapter 5), an implementation of our change distilling algorithm (see Chapter 4) to extract all source code changes from the revisions stored in the EVOLIZER RHDB. The resulting change types are stored in our EVOLIZER RHDB.

3. For each context and update change of invocations of a method, ZBINDER (Pinzger *et al.*, 2007) resolves the class name in which the method is implemented. An aggregation of invocation changes for each method is generated and stored in the EVOLIZER RHBD.

4. We rank the methods whose invocations are affected by context and update changes. We also extract change patterns among invocation changes and categorize them.

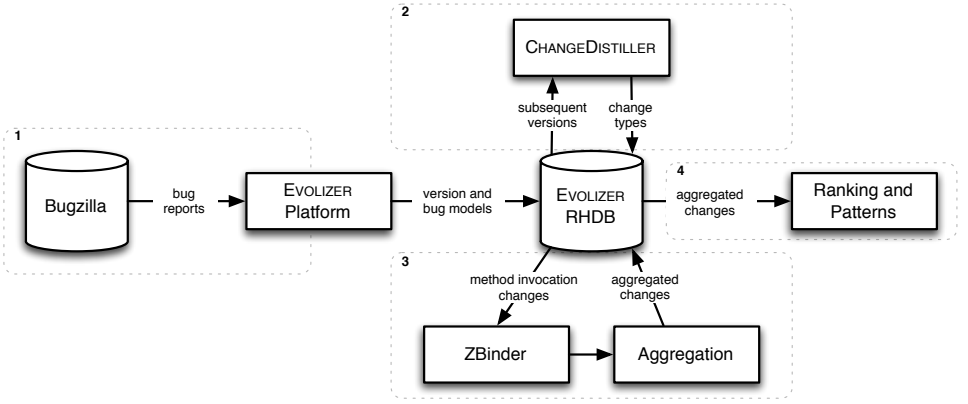


Figure 8.1: Process of data extraction and preparation

8.2.1 Versioning and Bug Data

To import bug reports from Bugzilla, we first parse each commit message to find indications for a bug fix. We use the strategy developed by Śliwerski *et al.* (2005b). Briefly, the strategy calculates two levels of confidence. First, at the syntactic level, the commit message is split into tokens and matched against regular expressions, such as bug numbers or bug related keywords. Second, at the semantic level, the link established from the syntactic analysis is validated. Factors such as the resolution, *e.g.*, `FIXED`, or the similarity between the commit message and the bug description are taken into account. When a link could be established, the bug report and the link are stored in the RHDB. For a detailed discussion of our integration into Evolizer we refer to (Jakob, 2007).

8.2.2 Change Extraction and Selected Change Types

Source code changes are extracted and classified by our change distilling algorithm (see Chapter 4). The classification is based on our taxonomy of source code changes which defines source code change types according to tree edit operations in the abstract syntax tree (AST) (see Chapter 3). In particular, our change distilling algorithm applies tree differencing pairwise on subsequent versions of ASTs of classes to extract the tree edit operations. We have implemented the algorithm in the Eclipse plugin ChangeDistiller (see Chapter 5) and it currently works for the Java programming language.

Because of Observation 8.2, we use two particular change types for ranking the method whose invocations are affected by context or update changes. We describe

them in the following.

Method invocation context change

Moving a particular method invocation into the then or the else-part of an if-statement is an instance of the change type *statement parent change* as defined in Chapter 3. The definition of this change type is as follows: The change applied to a statement is a statement parent change, if, and only if, the AST node of the statement is moved from its old parent node to a new parent node and the old parent node is not the new parent node. Consider the following parent change; a statement is moved into the then-part of an if-statement:

Old version:

```
public void foo() {
    aList.add(e);
}
```

New version:

```
public void foo() {
    if (!aList.contains(e)) {
        aList.add(e);
    }
}
```

When CHANGEDISTILLER extracts such a change it stores the string representation of the method invocation, *i.e.*, `aList.add(e);`, in which part the invocation was moved, *i.e.*, then-part, and the condition of the if-statement, *i.e.*, `!aList.contains(e)`.

For the ranking we focus on method invocation parent changes whose new parent is the then or the else-part of an if-statement. This filtering is also motivated by Observation 8.2. For the remainder of this chapter, we use the term *context change* to indicate a *method invocation parent change into the then or the else-part of an if-statement*.

Method invocation update

A particular method invocation update is an instance of the change type *statement update*. The definition of this change type is as follows: The change applied to a statement is a statement update, if, and only if, the value of the AST node of the statement, *i.e.*, the string representation of the statement, changes lexically. Consider the following update; the argument name has changed:

Old version:

```
public void foo() {
    aList.add(e);
}
```

New version:

```
public void foo() {
    aList.add(entity);
}
```

When CHANGEDISTILLER extracts such a change it stores the string representation of the method invocation before and after the update change was applied,

i.e., `aList.add(e)` and `aList.add(entity)`.

For the ranking we filter out method invocation updates that change the qualifier of the method invocation only. The reason is that variable renaming changes do not influence the call itself. We focus on the method invocation updates that either change the method name, arguments (*i.e.*, number of arguments or argument name), or both. For the remainder of this chapter, we use the term *update change* to indicate a *method invocation update*.

Aggregation of change types

To rank methods we first collect all existing method invocation context and update changes from the EVOLIZER RHDB.

Second, we split each method invocation into three parts: (1) class name of qualifier, (2) method name, and (3) number of arguments. In case an update change was applied to an invocation we use the invocation before the update. We resolve the class name of the qualifier with ZBINDER, a tool that resolves method invocations in incomplete source code models. Out of the three parts we concatenate a generalized method signature:

```
<class name>.<method name>(<number of arguments>)
```

For instance, the generalized signature of the call `result.add(e)` is `List.add(1)`.

Third, we aggregate those invocation changes that have the same generalized method signature. The rank is then calculated for each generalized method signature according to the set of aggregated invocation changes.

ZBINDER. To construct the generalized method signature, we need to resolve the class name of the callee. During the process of aggregating the change information, we deal with incomplete source code data. At the time a change is processed only the source revision in which the change happend is available. Thus, we need a tool to resolve method invocations in incomplete source code models. Dagenais and Robillard (2008) also dealt with incomplete source code models for recommending adaptive changes and used a similar approach.

We use ZBINDER Pinzger *et al.* (2007), a tool that handles unresolved calls in Java source code. It uses the Java Development Toolkit (JDT) of Eclipse to build the FAMIX model out of the provided Java source code. FAMIX is an independent source code meta model for object-oriented programming languages (Demeyer *et al.*, 2001). We sketch the workflow of ZBINDER briefly.

Whenever JDT binding resolution fails, ZBINDER gathers the name of the method, the number of arguments, and partial type information of the call receiving class as well as the argument types. This information is passed as input to four matching heuristics to find a best suitable method: (1) A method declaration with the same name, (2) the same number of parameters, (3) the same or a superclass

of the call receiving class, and (4) the same parameter types. If such a method declaration is found the method call relationship is established and added to the FAMIX model.

For all unresolved parts of the fully qualified name of the method, ZBINDER adds `<undef>` to the name. For instance, if the callee of the method could be resolved but not its package name, then ZBINDER returns the method name: `<undef>.IProgressMonitor.beginTask(*)`.

8.2.3 Method Ranking

We rank a method based on the number of context and update changes that were applied to its invocations. In addition, we consider changes that fixed a bug as more important than those that did not fix a bug and weigh them accordingly. To rank a method, we first calculate a score S for each method.

$$S = (w_{cc} + w_{uc}) \cdot nC,$$

$$\text{with} \quad nC = \frac{\text{\# context changes} + \text{\# update changes}}{\max(\text{\# context changes} + \text{\# update changes})},$$

$$w_{cc} = \text{percentage of context changes that fixed bugs}, \quad \text{and}$$

$$w_{uc} = \text{percentage of update changes that fixed bugs}$$

Methods whose invocations were changed frequently get a high score. The term nC is, therefore, the main term. The number of context and update changes are normalized by the maximum number of context and update changes over all methods.

We weigh the term nC with the sum $w_{cc} + w_{uc}$. The weight is used to obtain a high score for methods whose invocations have a medium number of context and update changes but a high percentage of context and update changes that fixed bugs. Our experience has shown that the percentage of context changes that fix bugs is significantly higher than of update changes. We, therefore, compute the two percentages separately for the weight.

After calculating the score for each method, we sort all methods in *descending* order according to the score, so that the method with the highest score gets the top ranking (*i.e.*, Number 1 is the highest rank). The higher its rank, the more affected by context and update changes are its invocations. With this ranking,

methods whose invocations have medium number of context and update changes but a high percentage of those changes that fixed bugs are ranked higher than methods whose invocations have a high number of context and update changes but a medium bug fix percentage.

8.2.4 Extracting Change Patterns

According to Observation 2.3, invocations of the same method and their context tend to be changed similarly. We group the changes into categories of context and update change patterns. The categories simplify the discussion of occurring change patterns in Section 8.3.

Context changes. We group the context changes (CC) into four categories. If the if-condition is composed out of more than one expression by AND (&&) or OR (||), we first split the if-condition into single expressions. We distinguish if-conditions that contain parts of the invocation and such that do not, *i.e.*, whether or not parts of the invocation are subject to check before calling the corresponding method.

CC.Qual *Qualifier appears in condition.* Example:

```
if (<qualifier> != null) {
   <qualifier>.add(e)
}
```

CC.Args *Arguments appear in condition.* Example:

```
if (<argument> instanceof IMember) {
   <qualifier>.add(<argument>)
}
```

CC.Both *Both, qualifier and arguments, appear in condition.* Example:

```
if (!<qualifier>.contains(<argument>)) {
   <qualifier>.add(<argument>)
}
```

CC.Other *Other; neither qualifier nor arguments appear in condition.*

For each method, we group the context changes of its invocations into those four categories and process their condition expressions to extract the change patterns.

Update changes. We group the update change (UC) into three categories. In the examples, we use the *Old* and the *New* of the invocation to highlight the change.

UC.Name *Method name changed.* Example:

Old `<qualifier>.add(<argument>)`
New `<qualifier>.remove(<argument>)`

UC.Args *Arguments changed.* Examples:

Old `<qualifier>.add(<oldArgument>)`
New `<qualifier>.add(0, <oldArgument>)` or
New `<qualifier>.add(<newArgument>)`

UC.Both *Both, method name and arguments, changed.* Examples:

Old `<qualifier>.add(<oldArgument>)`
New `<qualifier>.put(id, <oldArgument>)` or
New `<qualifier>.put(id, <newArgument>)`

For each method, we group the update changes of its invocations into those three categories and process their old and new version to extract the change patterns.

8.3 Experimental Results

In this section we describe our results of applying our ranking and change pattern extraction approach to the Eclipse software system. In Section 8.3.1 we present the experimental setup; in Sections 8.3.2 and 8.3.3 we evaluate the results and describe our findings.

8.3.1 Experimental Setup

We have chosen three core components of Eclipse to perform our experiment:

1. Eclipse Core (21 plugins from the Eclipse platform component; observation period: May 01 – Sep 07)
2. Eclipse JDT (17 plugins from the Java Development Tools component; observation period: May 01 – Sep 07)
3. Eclipse PDE (5 plugins from the Plugin Development Environment component; observation period: May 01 – Sep 07)

Table 8.3 summarizes these components. For each component we selected the main plugins without their test plugins. The CVS repositories of these components contain version histories of a time period of about 6.5 years. In total we populated our

database with about 172,000 Java source revisions comprising about 1.1 Million source code changes (instances of changes types).

Comp.	# source revisions	# linked fixed bugs	# bug fix revisions (%)	# changes	LOC first	LOC last
Core	15,454	1,470	4,495 (29%)	69,383	61,592	133,574
JDT	121,442	11,498	34,375 (28%)	904,786	420,233	974,006
PDE	35,137	1,680	5,468 (15%)	153,891	66,638	225,516
Total	172,033	14,648	44,338 (26%)	1,128,060	548,463	1,333,096

Table 8.3: Analyzed Eclipse Components (Comps.). [# source revisions] indicates the total number of revisions of Java files. [# linked fixed bugs] indicates the number of fixed non-enhancement bugs that could be linked to a revision. [# bug fix revisions] indicates the number of source revisions that are linked to a fixed non-enhancement bug. [# changes] indicates the number of source code changes that were applied during the period. [LOC first] and [LOC last] indicate the lines of code for the first and the last release of the component in the period

The bug-revision linking, described in Section 8.2.1, obtained 14,648 fixed non-enhancement bugs from Eclipse's Bugzilla[2] and linked them to 44,338 different Java source revisions. That means, 26% of all revisions are due to bug fixes.

According to Observation 8.2, context and update changes are responsible for fixing a significant amount of bugs. In Table 8.4 we list the corresponding numbers for Eclipse. Not all bugs are fixed with source code changes. Bug fixes also occur in other files, *e.g.*, `plugin.xml`, or affect licence terms. We use the term *source bug fix* for bugs that are fixed with at least one source code change.

Component	% source bug fixes	% context bug fixes	avg_{cc}	% update bug fixes	avg_{uc}
Core	80%	13%	2	27%	3.6
JDT	85%	15%	2.5	32%	2.5
PDE	86%	15%	2.3	41%	4.7
Total	84%	15%	2.4	33%	4.9

Table 8.4: Source code changes per bug. [% source bug fixes] indicates the percentage of bugs that are fixed by at least one source code change. [% context bug fixes] and [% update bug fixes] indicate the percentage of source bug fixes that contain at least one context or update change. [avg_{cc}] and [avg_{uc}] are the average numbers of context or update changes per context and update bug fix

[2]https://bugs.eclipse.org/bugs/

Overall, 15% of all source fixes are fixed with 2.4 context changes on average (avg_{cc}), 33% with 4.9 update changes (avg_{uc}). Given that we can extract over 120 different change types, we conclude that context and update changes are frequent bug fixes.

Next, we address our two research questions. With the ranking of methods we expect to highlight those methods whose invocations are significantly more affected to context and update changes than other methods. By analyzing the context and update changes among the invocations of the highest ranked methods we expect to reveal patterns of context and update changes.

We start by describing the results of the ranking. The ranking along with the scores and numbers used for this calculation are listed in Table 8.5. We list selected methods among the top ten ranked methods for each investigated Eclipse component. The complete list of the top ten ranked methods is presented in Appendix F.

8.3.2 Results of Method Ranking

In this section we address our first research question: Do methods exist whose invocations are significantly more affected by context and update changes than other methods? We separate the discussion for Java SE Development (JDK) and Eclipse related methods.

JDK methods. As we can see in Table 8.5 among all methods invoked in Eclipse, methods of the JDK are highly ranked. Invocations of `StringBuffer.append(1)` change often. In 35% to 50% of those cases, a condition check was inserted to fix a bug and in 20% to 47% the method invocation was updated because of a bug. Considering JDT, the number of `StringBuffer.append(1)` invocation context and update changes (2,541) is interesting. The second most method invocations affected by context and update changes are those of `Map.put(2)` with 712 changes—3.7 times fewer than the method with the most change-affected invocations. A search using Eclipse revealed that 2,263 references to `StringBuffer` are found in the last release of the 17 plugins of JDT comprising 5,921 (plus 1,852 potential matches) calls to `StringBuffer.append(1)`. For instance, code completion assistants and the compiler of JDT use `StringBuffer.append(1)` frequently. Similar to `StringBuffer`, `PrintWriter.println(1)` is used to generate information output. In PDE 808 calls to this method are found in the last release. PDE uses `PrintWriter` to generate `plugin.xml` files and predefined plugin websites.

The methods `ArrayList.add(1)` and `List.add(1)` are interesting. `ArrayList.add(1)` appears in the top ten list of Core (Rank 10) and of PDE (Rank 3); `List.add(1)` in the list of JDT (Rank 3) and PDE (Rank 7). In PDE 22% of all context and update changes of invocations of `ArrayList.add(1)` are due to a bug fix, for `List.add(1)` 53%. Since `List` is an interface and we can assume that the

use of a particular kind of list implementation does not influence changes to its methods, we determine that (at least in PDE) algorithms using a list data structure tend to change often and are affected by bugs. The methods `Map.put(2)` and `Hashtable.put(2)` belong to the category of data structure as well.

Eclipse methods. Besides the usage of the JDK, methods inside Eclipse also appear in the top ten rankings. `Assert.isTrue(..)` invocations are mostly used and affected by context and update changes in Core. We can also find them in JDT and PDE sporadically, but not to that extent as to appear in the top ten list. Asserts are used to check condition and throw an exception in case the condition evaluates to false. As listed in Table 8.5, 94% and 100% of all context changes of invocations of `isTrue(..)` are bug fixes. Misplaced asserts lead to a system halt and must be called with care.

Both JDT and PDE contribute to the UI of Eclipse. Therefore, methods of SWT[3] classes appear in the top ten list of JDT and PDE. Especially `*.setText(1)` invocations are frequently context and update changed.

Logging is used extensively in Eclipse. In Core and JDT the default logging occupies Rank 2, in PDE Rank 84. The reason for the high rank of using logging is that it has to be adapted whenever the source code for which it logs information changes.

Summary. We can confirm the first research question: Methods exist whose invocations are significantly more affected by context and update changes than other methods. These methods were listed by using our ranking approach.

[3]Standard Widget Toolkit, http://www.eclipse.org/swt/

Method (# methods)	Rank	Score	# context changes	# context fixes (%)	# update changes	# update fixes (%)
Core (1,162)						
`Assert.isTrue(2)`	1	0.95	17	16 (94%)	93	20 (22%)
`SB.append(1)`†	2	0.47	4	2 (50%)	66	27 (41%)
`<undef>.log(1)`	3	0.47	6	3 (50%)	69	24 (35%)
`Map.put(2)`	4	0.41	6	2 (33%)	55	31 (56%)
`Hashtable.put(2)`	5	0.29	20	20 (100%)	7	3 (43%)
`Assert.isTrue(1)`	6	0.27	1	1 (100%)	18	16 (89%)
`ArrayList.add(1)`	10	0.16	5	1 (20%)	28	13 (46%)
JDT (11,585)						
`SB.append(1)`†	1	0.74	581	230 (40%)	1958	669 (34%)
`<undef>.log(1)`	2	0.26	37	35 (95%)	351	273 (78%)
`List.add(1)`	3	0.21	178	82 (46%)	510	165 (32%)
`Map.put(2)`	4	0.21	98	34 (35%)	612	246 (40%)
`Button.setText(1)`	7	0.15	11	7 (64%)	428	99 (23%)
PDE (3,509)						
`PW.println(1)`‡	1	0.42	41	10 (24%)	490	87 (18%)
`SB.append(1)`†	2	0.30	78	27 (35%)	215	43 (20%)
`ArrayList.add(1)`	3	0.26	57	18 (32%)	204	45 (22%)
`Map.put(2)`	4	0.23	14	7 (50%)	136	44 (32%)
`Label.setText(1)`	5	0.18	7	3 (43%)	195	11 (6%)
`List.add(1)`	7	0.15	15	7 (47%)	64	34 (53%)
`Button.setText(1)`	10	0.11	4	1 (25%)	207	8 (4%)

†`SB` = `StringBuffer`
‡`PW` = `PrintWriter`

Table 8.5: Selected highly ranked methods for each component. The number inside the parentheses besides the method name indicates the number of parameters. [# context changes] and [# update changes] are the total number of context changes and update changes on invocations of the method. [# context fixes] and [# update fixes] indicate the number of context or update changes that fixed a bug on an invocation to the method. The methods are ordered descending by their rank. `<undef>` means that ZBINDER was not able to resolve the class name

8.3.3 Results of Change Pattern Extraction

In this section we address our second research question: Can we reveal change patterns among method invocations that are affected by context and update changes?

For each of the methods, we collected the context and update changes applied to their invocations. We grouped similar changes as described in Section 8.2.4. Table 8.6 lists the percentages of changes that were bug fixes as well as all other changes for context change categories; Table 8.7 for update change categories. Specialties are highlighted and discussed in the text.

We separate the discussion for Java SE Development (JDK) and Eclipse related methods.

Context changes

As we described in Section 8.2.4 context changes are grouped into the categories: (1) qualifier appears in condition (CC.Qual), (2) arguments appear in condition (CC.Args), (3) both, qualifier and argument, appear in condition (CC.Both), or (4) other checks are in condition (CC.Other).

Except for JDT, most of the context changes are in CC.Other. Calls of data structure methods of the Java library in JDT mainly are in CC.Qual or CC.Args.

JDK methods. The method `Map.put(2)` is the only JDK method that is not strictly in CC.Other. In Core and JDT `Map.put(2)` is in CC.Args as well. The conditions added before calling this method are verifying the validity of adding the argument to the map. Examples are **`null`** reference checks: `<argument> != null` or equality checks: `<argument> != superType`.

All other JDK data structure methods are in CC.Other, but have a number of CC.Qual and CC.Both patterns. For `List.add(1)` or `ArrayList.add(1)` the most frequently occurring qualifier condition is `<qualifier>.contains(..)`, it appears 29 times including negated (`!`) occurrences. In 86% of the cases it is checked whether the list contains the argument of the `add(1)` call: `<qualifier>.contains(<argument>)`. Nonetheless, checks for **`null`** references of arguments are also made for list data structures. Prominent examples for CC.Other change patterns are **`null`** reference or **`instanceof`** checks of other objects.

All context changes of the method `Hashtable.put(2)` in Core are bug fixes. The context changes in CC.Other category are adding condition checks whether a particular value is equal to a predefined default value.

The JDK method whose invocations have the most CC.Args changes is `System.arraycopy(5)`. The change patterns we found in this category check index pointers that are passed to `arraycopy`. A prominent example condition we found is: `++this.commentPtr >= <argument>`.

			CC.Qual		CC.Args		CC.Both		CC.Other	
Method	Rank	#	%n	%f	%n	%f	%n	%f	%n	%f
Core										
`Assert.isTrue(2)`	1	17	0	0	0	0	0	0	6	94
`SB.append(1)`†	2	4	0	0	25	0	0	0	25	50
`<undef>.log(1)`	3	6	0	0	0	0	0	0	63	38
`Map.put(2)`	4	6	17	0	17	17	0	0	33	17
`Hashtable.put(2)`	5	20	0	0	0	25	0	0	0	75
`Assert.isTrue(1)`	6	1	0	0	0	0	0	0	0	100
`ArrayList.add(1)`	10	5	13	0	0	0	0	0	75	13
JDT										
`SB.append(1)`†	1	581	0	0	4	3	0	0	55	37
`<undef>.log(1)`	2	37	0	5	5	62	0	7	0	21
`List.add(1)`	3	178	0	1	13	12	3	3	39	28
`Map.put(2)`	4	98	2	2	24	9	6	1	35	21
`System.arraycopy(5)`	6	124	0	0	32	34	0	0	26	8
`PR.handle(5)`‡	8	16	10	50	0	0	0	0	15	25
PDE										
`PW.println(1)`*	1	41	2	0	0	0	0	0	73	25
`SB.append(1)`†	2	78	0	2	7	5	0	0	63	24
`ArrayList.add(1)`	3	57	1	0	9	10	6	3	49	22
`Map.put(2)`	4	14	0	0	14	7	7	0	29	43
`List.add(1)`	7	15	0	0	6	0	6	0	35	53

†`SB` = `StringBuffer`
‡`PR` = `ProblemReporter`
*`PW` = `PrintWriter`

Table 8.6: Context changes split into categories of the selected methods. [%f] indicates the percentage of changes that fixed bugs. [%n] indicates percentage of non-fix changes. The context change categories are: (1) qualifier in condition (CC.Qual), (2) arguments in condition (CC.Args), (3) both qualifier and arguments in condition (CC.Both), or (4) other condition checks (CC.Other). Specialties are highlighted and discussed in the text

`StringBuffer.append(1)` as well as `PrintWriter.println(1)` are mainly in CC.Other.

Eclipse methods. Most of the context changes are in CC.Other. The methods `ProblemReporter.handle(5)` and `log(1)` used in JDT, however, are exceptional.

Logging in JDT is in CC.Args. A fix example for logging is checking whether the argument exists: `!<argument>.isDoesNotExists()`.

The method `ProblemReporter.handle(5)` is in CC.Qual—the only method whose invocation context changes are in CC.Qual in all of the three top ten list. For all context changes a condition to verify the Javadoc visibility is added before calling this method.

Update changes

As we described in Section 8.2.4 update changes are grouped into the categories: (1) method name changes (UC.Name), (2) arguments change (UC.Args), or (3) both, method name and arguments, change (UC.Both).

Update changes of method invocations in the top ten list mainly are in UC.Args as one can see in Table 8.7. Method name changes appear rarely, an indication that, for instance, data structures are seldom replaced. Moreover, 80% of all methods are in UC.Args, meaning that a method invocation is seldom updated due to fix a bug. As in the previous sections, we start with discussing JDK methods.

JDK methods. The arguments of invocations to data structures, such as lists or maps, are updated frequently. A significant amount of argument changes in JDT and PDE are *replace query by temp variable*. Similarly, in PDE the variable `model` is used to store any kind of model in a list. In addition, in JDT as well as in PDE other refactorings took place which induced changes of the arguments: In PDE a utility class was introduced; in JDT a prominent class was renamed, *i.e.*, `JavaRefactoringDescription` was renamed to `JDTRefactoringDescriptor`.

Since JDT uses `StringBuffer` intensively, we found interesting patterns of argument changes. Most frequently are patterns that correct the appending of single characters. For instance, `append(',')` replaced `append(",")` or `append(";")`. Other characters are replaced by constants: `append('␣')` by `append(Util.bind("disassembler.space"))`. Moving constants is also a frequent pattern. Finally, invocations to `Util.bind("disassembler.space")` were replaced by `Message.disassembler_space`. In the course of the Eclipse project, the class `Message` together with a `messages.properties` file replaced most of the string constant initializations. We found similar character replace pattern in Core and PDE—but they are not as frequent as in JDT.

Method	Rank	#	UC.Name %n	UC.Name %f	UC.Args %n	UC.Args %f	UC.Both %n	UC.Both %f
Core								
`Assert.isTrue(2)`	1	93	0	0	78	22	0	0
`StringBuffer.append(1)`	2	66	0	0	59	41	0	0
`<undef>.log(1)`	3	69	0	0	65	35	0	0
`Map.put(2)`	4	55	0	0	44	55	0	2
`Hashtable.put(2)`	5	7	0	14	57	29	0	0
`Assert.isTrue(1)`	6	18	0	0	11	89	0	0
`ArrayList.add(1)`	10	28	0	0	54	46	0	0
JDT								
`StringBuffer.append(1)`	1	1958	0	0	66	34	0	0
`<undef>.log(1)`	2	351	5	32	17	46	0	0
`List.add(1)`	3	510	0	1	67	30	1	2
`Map.put(2)`	4	612	0	0	59	40	0	0
`System.arraycopy(5)`	6	506	0	0	81	19	0	0
`ProblemReporter.handle(5)`	8	371	0	0	77	23	0	0
`VPS.println(2)`†	9	333	0	0	0	100	0	0
PDE								
`PrintWriter.println(1)`	1	490	0	0	80	17	2	1
`StringBuffer.append(1)`	2	215	0	0	80	20	0	0
`ArrayList.add(1)`	3	204	0	0	72	21	6	1
`Map.put(2)`	4	136	0	0	68	32	0	0
`List.add(1)`	7	64	0	0	45	53	2	0

†`VPS` = `VerbosePacketStream`

Table 8.7: Updates split into categories of the selected methods. [%n] indicates percentage of normal changes. [%f] indicates the percentage of changes that fixed bugs. The update categories are: (1) method name changed (UC.Name), (2) arguments changed (UC.Args), or (3) both, method name and arguments, changed (UC.Both). Specialties are highlighted and discussed in the text

Instead of the `StringBuffer` to prepare strings, PDE writes them directly with a `PrintWriter`. Update changes on calls to `PrintWriter.println(1)` are comparable with update changes to `StringBuffer.append(1)`. We found patterns where either spaces to make indents are replaced by tab (`\t`) or where the number of spaces were increased or decreased in almost all `PrintWriter.println(1)` invocation update changes.

Eclipse methods. We first discuss update change patterns in Core. The signature of `Assert.isTrue(2)` is `Assert.isTrue(boolean, String)`. About 18% of update changes on `Assert.isTrue(2)` calls replace the boolean argument with `false`: `isTrue(false, <oldArgument>)`. The same string constant initialization mechanism, as we described above, is used with `Assert.isTrue(2)` calls and, therefore, a significant amount of string arguments are replaced accordingly.

Consider the `log(1)` call in JDT. It is in UC.Name—the only method whose invocation changes are in UC.Name. We found a two step `log(1)` update pattern. First, 156 calls resembling `*.log(x.getStatus())`, where `x` stands for any object, are changed into `*.log(x)`. Second, about 71% of those new invocations are changed into `*.logIgnoringNotPresentException(<oldArgument>)`.

The method `VerbosePacketStream.println(2)` in JDT is a special UC.Args candidate. Its invocations were updated 333 times, all due to fix a bug. Compared to the other listed methods in Table 8.7 this method is unique. All these update changes can again be separated in a two step pattern. First, the string constants are replaced by the `Message` class and `message.properties` construct. Second, the constant pointing to the `Message` class is replaced by an ordinary string: All calls similar to the following expression

```
println(*Messages.getString("<identifier>"), <argument>)
```

were changed into

```
println(*Messages.<identifier>, <argument>)
```

and afterwards into, for instance,

```
println("Classes count:",<argument>).
```

Summary. We can confirm the second research question: We can reveal change patterns among invocations that are affected by context and update changes.

8.3.4 Summary of Experiment

We conclude this section with a summary of our experiment. In the next section, we discuss the usefulness of the results for our long-term goal.

Method ranking. The ranking of the methods of three core components of Eclipse shows that invocations to string handling, output generation, and data structure

methods in JDK are highly ranked. In particular, invocations of `StringBuffer.append(1)`, `PrintWriter.println(1)`, and `List.add(1)` are used, changed, and involved in many bug fixes frequently. Methods within the Eclipse framework, such as `ILog.log(1)` or `Button.setText(1)`, appear also in the top ten list.

Context change patterns. Context change patterns are mostly in CC.Other. We also found CC.Args and CC.Qual change patterns in all three components. It is surprising though that the most common change is an insert of a `null` reference check before a method call. The components JDT and PDE share similar condition check patterns, mainly in calling JDK related methods. For instance, containment or object properties checks are added multiple revisions after an invocation to `List.add(1)` is inserted.

Update change patterns. Update change patterns are mostly in UC.Args. The only exception are logging invocations in JDT. They are also in UC.Name. Refactoring related update changes appear more frequently for invocations of JDK methods than for methods of Eclipse. For instance, query-arguments in those invocations are replaced by temp variables.

The most prominent update change patterns are (1) replacing string constants with the `message.properties` mechanism along with the `Message` class of Eclipse, and (2) replacing characters with strings in `StringBuffer.append(1)` invocations.

8.4 Discussion

Our ranking revealed interesting methods whose invocations are affected by context changes and update changes. In particular, we did not expect to find an overlap in using data structure methods. On the other hand, the frequency of patterns found in method context changes and update changes are rather low, although we investigated a corpus of change histories with more than a million source code changes. The invocations of methods in the top ten list have rarely over 100 context changes—in only three cases out of 30. In this section, we discuss whether the ranking we used is good enough to raise the awareness for which method a developer has to pay attention, and whether the change patterns we revealed are meaningful enough for our long-term goal, *i.e.*, to provide change suggestions during development to reduce the number potential bugs. We also report on additional information we have considered to include for future improvements. We conclude by discussing threats to validity of our approach.

8.4.1 Assessment of the Method Ranking

We rank a method based on the frequency of context and update changes that were applied to its invocations. In addition, we consider changes that fixed a bug as more important than those that did not fix a bug and weigh them accordingly. Methods whose invocations changed most often and were involved in bug fixes are among the top ten ranked methods (see Table 8.5).

Invocations to data structure and string handling methods of JDK are affected by context and update changes in all three core components. We did not expect this finding, since those methods are frequently used in many Java software projects. The focus on context changes in our ranking approach is a reason for their high rank. For instance, a list data structure is used in an algorithm. The algorithm expects only to add results to the list that have certain properties. To verify these properties, condition checks are necessary. Because of that, the invocation to add the result is moved inside a corresponding if-statement. It is worth ranking the method `List.add(1)` highly because the rank points out that one has to pay attention when using this method.

Bug severity and priority. Typically, each bug report is classified with a severity and a priority. With this information we can additionally weigh bug fixes. A method is then ranked higher if its invocations were often context and update changed due to fixing severe or high priority bugs. In addition, we can weigh methods whose invocations changed recently more strongly than invocations that did not change for a long time. However, we claim that, at least in Eclipse, the bug classification will not change the ranking significantly since about 90 percent of all bugs are classified as *medium* (*i.e.*, priority *P3*, severity *normal*).

Time dimensions. Time dimensions such as the recency of changes could improve the ranking. For instance, the information that a certain kind of method invocation was changed recently is of interest: When adding such a method invocation, we can suggest up-to-date changes accordingly.

8.4.2 Assessing the Change Patterns

In the long-term we expect to leverage context and update change patterns to support developers by suggesting method invocation changes. The suggestions shall be integrated into the Eclipse IDE and recommend the changes when a developer inserts a certain method invocation. Assume a developer adds the call `result.add(e.getMessage())` to a method. By inspecting the call and the parent node in which the invocation is inserted we can provide the following information:

1. The rank of the method whose invocation she is adding represents the risk of potential additional changes; we say: "Be aware that the rank of the method `List.add(1)` is high (Rank 3)."

2. The stored change patterns of the method invocation can provide change suggestions if requested. We list the change pattern categories that occurred the most and give a set of context and update change recommendations. We recommend those changes that were applied frequently and were often involved in bug fixes. For `result.add(e.getMessage())` the recommendation could then be:

   - Context changes in CC.Qual as well as CC.Both and we suggest to:
     - Add an if-statement with the condition `e.getMessage() != null` and call the method inside the then-part of the if-statement.
     - Add an if-statement with the condition `!result.contains(e.getMessage())` and call the method inside the then-part of the if-statement.
   - Update changes are in UC.Args and we suggest to:
     - Replace query by temporal variable:

       ```
       String message = e.getMessage();
       result.add(message);
       ```

With the provided support, a developer first sees the potential context and update change risk the method invocation has. According to the change pattern category she can decide whether category is relevant for the method invocation and can apply the recommended change automatically.

The assessment of the extracted change patterns is whether the support, as described in the sample scenario, is valuable and feasible with the data gained; it is, but still limited. We discuss the reasons for context and update change patterns separately.

Context change patterns. The patterns we extracted from the invocations of the top ranked methods are indeed relevant. But we expected to gain a higher frequency of particular patterns. Context change patterns in one of the four categories rarely exceed an amount of seven percent. In particular, most context changes are in CC.Other. For instance, consider the `StringBuffer.append(1)` method whose invocations have heavily changed in JDT—string buffers are used for code completion suggestions. Decisions whether a certain string is added to the buffer for a completion suggestion are made upon general user preferences or object states which are not related to the `StringBuffer.append(1)` invocation itself. How appropriate change suggestions can be extracted in this case is subject of future work.

In addition, the patterns selected for the suggestions should be preprocessed with more detailed information about the call. For instance, if the type of the argument does not provide methods that are called in the condition of the pattern candidate, the recommender should not suggest the corresponding context change.

Update change patterns. Update change patterns appear more frequently than context change patterns. Switches in mechanism, such using strings via `Message` classes, are recognizable and can be used to suggest changes directly. But we have to filter the suggestions in future, too. Assume an invocation of `Map.put(2)` is added to PDE. Since the rank of this method is high, suggestions for update changes are made. These update changes have to be filtered according to the types of the arguments of the inserted call. For instance, a switch to the `Message` class for strings, mentioned earlier, should only be suggested, if one of the arguments is of type `String`.

Additional change information. For a recommendation system we plan to include additional change information. So far, we focused on single method invocations. We intend to extract nested method calls as well. Moreover, function invocations, *i.e.*, methods returning values, are not yet considered. Suggestions of postcondition checks are valuable as described by Williams and Hollingsworth (2005).

To conclude, the change pattern results that we described in this chapter show that recommending additional changes when adding a method invocation is valuable and feasible, but needs to be refined and augmented with additional information.

8.4.3 Threats to Validity

There are four major threats to the validity of this work.

Systems examined might not be representative. We examined three components from Eclipse. It is, therefore, possible that we have chosen an unrepresentative set of systems for our study. However, Eclipse is a well-known system in the software evolution research community and often used as a case study. Moreover, it has a rather long version history (over six years), provides an issue tracking system whose reports can be linked to revisions adequately, is developed all over the world, and is big enough to have diversity of development and source code patterns. The number of developers of the three components are as follows: Core has 47, JDT 55, and PDE 23 authors. Consider the overlap of developers: Core and JDT have 19 common developers, Core and PDE have 10 common developers, and JDT and PDE have 7 common developers. According to these numbers, we have chosen a set of systems that share programming experience, which is appreciative

for finding common patterns, as well as we have a certain amount of distinction in development, which is appreciative for excluding bias. As our dataset increases, the severity of this threat will diminish.

Systems are all open source. The Eclipse project is not strictly open-source, the main developers are employed by IBM, but the source code is free and individuals can contribute to the project. The main issue when using open-source projects as case studies is the incomplete bug fix data. We must not assume that the links between bug reports and revisions in CVS are complete. It is still up to the developer whether or not she adds the bug number to the commit message. However, for our study we were able to link 14,648 reports of non-enhancement bugs to 44,648 revisions—a quarter of all revisions. To get more complete data we either have to find an open-source project having always bug report numbers in the commit message or find a commercial software system with a well defined development process.

A minor issue when using open-source projects is that the development methodology is not representative. It is possible that deadline pressures are stronger in commercial projects than in open-source projects, which might have an effect on the results of the ranking. Eclipse has a well defined deployment process including milestone and release deadlines.

All systems are written in Java. Extracting source code changes on the AST level requires a complete programming language parser. As a result CHANGEDISTILLER currently supports the Java language. Systems in other programming language may have different change patterns. However, we claim that the investigation of the changes on invocations to methods is independent from the programming language because on the statement level, programming language of the same paradigm are similar.

Incomplete source code model. We used ZBINDER to resolve method calls of incomplete source code models. Since ZBINDER relies on heuristics we resolved 90 percent of all method calls used for our study. We had about 15,000 method invocations to investigate—enough to evaluate our ranking and find patterns among the changes.

8.5 Résumé

In this chapter we investigated whether methods exist whose invocations are significantly more affected by context and update changes than other methods and

whether we can reveal change patterns among those invocation changes. We developed an approach that ranks how often context and update changes were applied to invocations of a particular method and whether these changes were bug fixes. In addition, we extracted patterns of context and update changes to assess whether they can be used to provide valuable change suggestions.

We performed experiments on three core components of Eclipse: Core, JDT, and PDE. We applied our approach to rank the methods called in Eclipse and extracted change patterns. The results showed that:

- Invocations to data structure and string handling methods of JDK are frequently used, changed, and involved in many bug fixes. The corresponding methods are therefore among the top ten ranked methods in all three investigated Eclipse components.
- We found relevant invocation change patterns but we expected a higher frequency. The most common context change is the `null` reference check before calling a method. In addition, `List.add(1)` invocations are frequently moved into the then or else-part of if-statements that have the condition `!list.contains(<argument>)`.
- Update patterns appear more frequently than context change patterns. We found that (1) query arguments of many invocations of JDK methods were replaced by temp variables, (2) string constants were replaced with the message mechanism, and (3) character arguments were replaced by string arguments.

In this chapter we have shown that the analysis of changes on invocations of methods highlights those methods whose invocations are affected by context and update changes. The found patterns among these invocation changes can be leverage to provide suggestive feedback for developers. We therefore accept Hypothesis H2c, and we regard the third part of Research Goal G3 as fulfilled.

IV

Retrospection

Contributions to Software Engineering

9

In his famous book, *The Mythical Man-Month*, Frederick Brooks states that software engineering is among the most complicated engineering disciplines because we build software systems that are composed of different interlocking concepts (Brooks, 1995). In addition, software systems can grow in size beyond many million lines of code, which a single person or even a group of persons cannot overlook. Creating software has always been complex, as the attendees of the first conference on software engineering in 1968 have already discussed (Naur and Randall, 1968). The problems they were dealing with were similar to the ones we have nowadays:

> "[..] poor performance, poor design, instability, and mismatching of promise and performance. [..] Our problem has arisen from a change of scale which we do not yet know how to reduce to alphabetic proportions."
>
> –A. J. Perlis, 1968, Keynote in (Naur and Randall, 1968)

They have already spoken about the *software crisis* (Osterweil, 2007). As a consequence, software engineering became a research discipline: Only three years after the first NATO conference on software engineering the well-known *International Conference on Software Engineering* (ICSE) emerged.

Software engineering research is driven by a big community nowadays. One of the major goals is to develop methodologies, techniques, and tools to ease the development and maintenance of software. The contributions of these activities are manifold. Just think of how the way we develop has changed in the last decades: New programming paradigms along with programming languages and integrated

development environments that support *debugging* source code evolved during that time.

According to Osterweil (2007), software engineering knows two research characteristics: The *problem-solving-oriented* and the *curiosity-driven* research. Apparently, the problem-solving-oriented research is omnipresent. The advantage of this ubiquity is that we are able to build large scale software systems. The problem-solving-oriented research must continue to drive us but we should also learn the *nature* of software engineering and, therefore, let us drive by curiosity. Curiosity is a common factor that drives natural sciences. We are sure that curiosity will lead to new methodologies and technologies to ease software engineering.

Software evolution analysis is a discipline of software engineering research. Thus, we expect from software evolution analysis to contribute to software engineering. In particular, we expect that analyzing *change distilled data* of software systems permits us to obtain insights of the nature of software engineering in general and software development in particular. We are convinced that we can develop techniques to facilite software development by leveraging these insights.

Since we contribute to software evolution research we also contribute to software engineering. The two research characteristics of software engineering are represented in the two dimensions of analyzing software evolution, as discussed in Chapter 1. Understanding software evolution is driven by curiosity research and the support of software evolution is driven by problem-solving-oriented research. In this chapter, we summarize our contributions with respect to the two perspectives. We describe the curiosities that led us to contribute to understanding software evolution and report on the corresponding empirical results. We also discuss for which tasks we can support developers by leveraging our results.

The research for this dissertation was mainly curiosity-driven. We have started with curiosity questions, *i.e.*, with one of our hypotheses (see Section 1.3.1). The tasks, for which we can contribute to a solution, were discovered during the analysis of our findings.

The change distilling algorithm indirectly contributes to the understanding and support of software evolution. It is our foundation of the software evolution experiments we conducted during the course of this dissertation. Without the change distilling algorithm, and CHANGEDISTILLER respectively, the experiments would not have been possible.

9.1 Understanding Software Evolution

The findings of our three software evolution experiments contribute to the understanding of software evolution.

9.1.1 On the Commenting Process in Software Projects

As code is read more often than written (Goldberg, 1987) comments are essential to understand source code. They also facilitate the introduction of the source code for developers unfamiliar with a software project. Developers are aware of the importance of comments but also neglect to maintain existing comments. Possible reasons are deadline pressure or carelessness (Lakhotia, 1993).

We assumed that we can learn from the quantitative examination of comments and their association to source code to understand the commenting process. We were eager to find out whether an analysis of the co-changes between comments and source reveals whether comments are kept up-to-date or re-documentation is common. We supposed that comment changes are mostly due to a source code changes. But, we were almost certain that a comment is seldom adapted to the source code in the same revision because of Lakhotia's (1993) finding. We expected that comments are rather adapted retroactively.

We developed an approach to associate comments with source code entities and conducted three empirical experiments to pursue our curiosity (see Chapter 6). Our findings revealed interesting insights:

1. The growth factor of source code and comments are similar over time in all investigated software systems. But this does not directly mean that newly added code is well commented because half of the investigated systems have a commented source code proportion of less than 50 percent. It rather means that the ratio of comments to source code remains stable.

2. The type of source code entity highly influences whether the entity is commented or not and there is also a partial order in the likeliness of whether a certain entity gets commented. Classes are more often commented than methods and attributes. If-statements and loops are more often commented than method invocation statements or other simple statements.

3. Over 50 percent of comment changes are induced by source code changes. For six out of eight investigated systems over 90 percent of these co-changes are applied in the same revision.

Finding 1 confirmed partly our expectation that source code is barely commented. That means, the proportions of lines of code that are commented is below 50 percent. Although Finding 2 showed that developers are aware of the importance of commenting higher level scopes, it also showed (see Figure 6.4) that higher level scopes—especially declarations—are not sufficiently commented. JFreeChart was definitely an exception in this experiment.

To get a more detailed look on the interaction between comment and source code changes, we analyzed Finding 3. The immediate adaptation of the comment

in the method body was against our expectations. On the other hand, higher level scopes do get re-documented in half of the investigated software systems.

Our findings can be used to qualify the commenting process of a software system. We can show how lines of comments and source code grow to indicate whether developers are commenting their code. The second experiment reveals whether source code is cleaned from dead code and whether developers are aware of the importance of higher level comments. In case this cannot be shown by our experiments a training for the developers can be appropriate. The last experiment shows the timeliness of the comments and indicates whether the commenting process is cost intensive. The former is important to ensure a certain level of comprehensibility for project newcomers. Because re-documentation requires more time than the immediate modification of comments, the third experiment may provide a basis to decide a change in the commenting process.

9.1.2 On the Nature of Change Types Applied Together

Certain source code changes are mostly applied together. For instance, a parameter renaming impacts all statements that access the parameter inside the method body. The statements have to be adapted to the preceding change. We were curious whether change type patterns exist that are beyond common refactoring patterns as described by Kim *et al.* (2007a,b).

We developed an approach to discover change type patterns by using hierarchical agglomerative clustering (see Chapter 7). We applied the clustering on the entire time frame and on quarterly periods of the historical data of three software systems. We distinguished therefore between *global* change type patterns, *i.e.*, patterns that appear during the entire history, and *local* change type patterns, *i.e.*, patterns that only appear during certain time periods (*e.g.*, yearly quaters). We expected that local change type patterns reveal inconsistent changes; changes that normally are not applied this way. For instance, assume exceptions are handled by catching them and re-throwing a system-defined exception. Inserting exception handling then leads to the change type pattern {try statement insert, catch clause insert, throw statement insert}. Assume further that during a certain time period, exception handling was differently inserted, *e.g.*, without a throw-statement. By detecting such inconsistent changes we assumed to find indications for bugs or developers who changed the source code unaware of certain guidelines.

The experiments showed patterns telling us something about the nature of change types applied together, but nothing that we had expected. We rather found that change type patterns reveal differences in using exception flow and that certain control flow changes indicate source code cleanup activities. Moreover, certain API convention cleanups are spread over the entire history of a software system. Whether those cleanup activities are due to an inconsistent use of coding guidelines or due to frequent modification of them remains unclear. We also made the

experience that in case of mixing feature development with code cleanup, a pattern extraction is not feasible with our approach. However, we can either show that coding guidelines are not followed by developers, that newcomers might not be appropriately trained, or that coding guidelines are frequently modified. This finding is relevant because unnecessary changes are costly and can be avoided.

9.1.3 On the Nature of Method Invocation Changes

Predicting error-prone software modules and, more recently, bug prediction research exists for a long time. In the seventies McCabe (1976) or Halstead (1977) have already proposed software measures to point to error-prone (*i.e.*, complex) modules. Nowadays more sophisticated approaches exist that also take historical data of software systems into account. We refer to (Kim *et al.*, 2008) for an interesting survey.

Similar to Kim *et al.* (2006b), we were curious about finding change patterns that fix bugs. During this investigation we observed that several bugs are fixed with fewer than four source code changes (instances of change types), such as, two *statement inserts* and one *control structure condition expression change*. In addition, a significant amount of these bugs were fixed with the change types *method invocation statement parent changes* and *method invocation statement updates*. Based on these observations we assumed certain methods exists whose invocations are significantly more affected by these two change types than of other methods (see Chapter 8). Furthermore, we expected to find out whether the changes applied to these invocations are similar, thus form change patterns.

The results that we obtained by analyzing the method invocation changes in the Eclipse project were promising. Interestingly, method invocations to the JDK library changed most often and were involved in many bug fixes. We also showed that the changes can be grouped to patterns because similar context and update changes were applied to the invocations. Therefore, the change patterns and the awareness under which conditions the change patterns were applied will contribute to avoiding future bugs. For instance, in our experiment with the Eclipse project a significant amount of `List.add(1)` method calls were moved into an if-statement with `!<qualifier>.contains(<argument>)` as the condition. For Eclipse, we can then define that for each insert of the call to the `List.add(1)` method the developer has to consider whether a corresponding check might be appropriate. How such a consideration can be supported during development we discuss in Section 9.2.3.

9.2 Supporting Software Evolution

During our curiosity driven research to contribute to understanding software evolution, we learned how we can benefit from our results to support software evolution. We consider the support of software evolution with support tools that are integrated into development environments such as Eclipse. Such an integration provides a closed feedback loop: (1) Historical data of specific development processes are collected in the development environment, (2) empirical approaches will be automated on this data, and (3) rules as well as recommendations will emerge from this data to effectively support developers (Zeller, 2007). To give an idea how such an integration will support developers in their daily business we provide several scenarios.

9.2.1 Supporting Adaptive Comment Changes

Comments have to be kept up-to-date to reduce the cost of re-documentation. We propose to integrate a support tool that suggests when to adapt comments to source code changes. Such a tool will directly leverage our extracted change types from a software systems.

Scenario. Assume a developer makes source code changes in the declaration and the body of a method. For instance, she changes the type of a parameter and makes several changes inside an if-statement accordingly. Assume further that the method has a Javadoc and that the if-statement is described by a preceding comment. The statements inside the if-statement are not commented.

After applying the changes, the tool checks whether the Javadoc was adapted to the parameter type change. A simple equality check of the current version and the previous version of the Javadoc is sufficient. In case the Javadoc did not change yet. The tool suggest to adapt the Javadoc. Furthermore, the tool collects the change types applied inside the if-statement and sums up their change significance levels. In case the total change significance level exceeds a predefined threshold it checks whether the comment describing the if-statement has already been adapted to the source code changes. If not, it suggest the developer to do so. In both cases the change significance levels will also be used as a confidence level for the suggestions.

The developer can then decide whether or not she should apply the comments. If she decides to postpone the adaptation, the tool can at least tag both comments for later consistency approval and adaptation.

9.2.2 Supporting Consistent Changes

The results of our change type pattern experiment showed that cleansing of source code is a significant development activity. A tool that learns coding guidelines from past change type patterns can monitor applied changes and suggest appropriate changes in case of a guideline violation. We consider two scenarios in which such a tool can be relevant.

Scenario 1: Newcomer. Assume a new developer joins a software development team. Assume further that the team is under deadline pressure and has only very limited time left to introduce the newcomer to the software system and their coding guidelines—corresponding documentation does not exist (yet). The newcomer starts with her programming task right away. Since she is not aware of the coding guidelines, she has plenty of questions during development: Do they use single or multiple exits? Does a common mechanism to handle exceptions exist? How should if-statements be constructed? A look into different code snippets does not answer her questions because her teammates do not change consistently. Two possible solutions comes into the developers mind. First, she develops as she is used although she might violate some guidelines, or, second, she asks someone. The developer decides for the former because she does not want to bother their teammates.

We suggest a third solution. By automatically learning from the past change type patterns, a tool can act in two different ways to support the newcomer. First, it provides coding suggestions by acting as a search engine. For instance, the newcomer can ask what is the most common way to handle exceptions. Because of the past insert and deletes of exception handlings the tool can provide appropriate code snippets.

Second, the tool can check the changes that the newcomer made. For instance, single exit is mainly used in the software project. In case the newcomer adds more return statements, the tool signals a warning, tells what might be wrong, and suggest a corresponding change. The change will then be automatically applicable.

Scenario 2: Warning a developer. The tool described for Scenario 1 can also be used to warn developers. The tool warns such developers that they might have violated a coding guideline and suggest a corresponding adaptation of the source code.

For both scenarios the proposed tool can limit the effort of a late adaptation of the source code to coding guidelines. It can also limit the negative effect of frequent guideline modifications but it cannot avoid them. Defining clear and strict coding guidelines at the beginning of a software project is always preferable over their late introduction.

9.2.3 Supporting the Use of Methods

Our observations showed that a significant amount of bugs are fixed with similar source code changes, especially on method invocations. By ranking methods whose invocations are affected by context as well as update changes and extracting change patterns accordingly, we aim at providing additional change suggestions to reduce the number of future bugs. For that, a corresponding tool suggests changes when a developer inserts a certain method invocation.

Scenario. Assume a developer adds the call `result.add(e.getMessage())` to a method. By inspecting the call and the parent node in which the invocation is inserted we can provide the following information:

1. The rank of the method whose invocation she is adding represents the risk of potential additional changes; we say: "Be aware that the rank of the method `List.add(1)` is high (Rank 3)."

2. The stored change patterns of the method invocation can provide change suggestions if requested. We list the change pattern categories that occurred the most and give a set of context and update change recommendations. We recommend those changes that were applied frequently and were often involved in bug fixes. For `result.add(e.getMessage())` the recommendation could then be:

   - Context changes in CC.Qual as well as CC.Both and we suggest to:
     - Add an if-statement with the condition `e.getMessage() != null` and call the method inside the then-part of the if-statement.
     - Add an if-statement with the condition `!result.contains(e.getMessage())` and call the method inside the then-part of the if-statement.
   - Update changes are in UC.Args and we suggest to:
     - Replace query by temporal variable:

       ```
       String message = e.getMessage();
       result.add(message);
       ```

With the provided support, a developer first sees the potential context and update change risk the method invocation has. According to the change pattern category she can decide whether the category is relevant for the method invocation and can apply the recommended change automatically.

9.3 Résumé

Software evolution analysis is a discipline of software engineering research. The two dimensions, *understanding* and *support*, are related to the *curiosity-driven* and *problem-solving-oriented* perspectives of software engineering research. Because of that, we contribute to both perspectives:

- *To the understanding of software evolution* by empirically analysing the commenting process in software projects, by investigating the nature change types applied together, and by investigating the nature of method invocation changes.
- *To the support of software evolution* by proposing tools that leverage our extracted source code changes. These tools suggest adaptive comment changes, suggest consistent changes, and support the use of methods.

With the discussion in this chapter we have argued for having found enough evidence for our thesis. That means, we provided evidence that extracting fine-grained source code change from the history of a software system contributes to the understanding and to the support of software evolution. We, therefore, regard the thesis as verified.

V

Closing

10 Conclusions

One effective way to limit the negative consequences of *ignorant surgery* is by putting the *source code change* in the center of the analysis. With change distilling, we presented an approach that enriches software evolution analysis with source code change histories. By defining and classifying source code change types with respect to tree operations in the abstract syntax tree (Fluri and Gall, 2006) we enabled the extraction of fine-grained source code changes with our change distilling algorithm (Fluri *et al.*, 2007a) and its implementation, the ChangeDistiller.

As shown by our experiments we contribute to the understanding and the support dimensions of analyzing software evolution. The relation of comment to source code changes enabled us to investigate whether comments and source code co-evolve (Fluri *et al.*, 2007b, 2008a). We found out that the adaptation of method body comments to source code changes happen in the same revision whereas API comments are adapted several revisions after the source code change happened, *i.e.*, they got re-documented. The comment and code co-change analysis can reveal whether source code is adequately commented and whether the comments are kept up-to-date with the source code. Both insights are important to ensure a certain level of comprehensibility for project newcomers. Change types and their change significance levels indicate when a developer might adapt a comment.

Agglomerative hierarchical clustering of change types revealed that certain control flow changes are due to particular source code cleanup activities, that exception flow is used differently in system parts, and that API convention changes are spread over many releases (Fluri *et al.*, 2008b). Change type patterns can show that coding guidelines are not followed by developers, that newcomers are not

appropriately trained, or that coding guidelines are frequently modified. Such findings are relevant for project managers because unnecessary changes are costly and can be avoided. Change type patterns can also support project newcomers to apply coding guidelines adequately.

By analyzing the change history of method invocations we enabled the ranking of methods whose invocations are affected by context and update changes (Fluri and Gall, 2008). By further extracting patterns of these invocation changes we assessed how developers can be supported with lists of potential change suggestions to reduce the number of recurring bugs.

The results of our three software evolution experiments provided enough evidence that the analysis of change types helps in understanding software evolution and provides means to support developers in their daily work.

10.1 Acceptance of Hypotheses

For the fulfillment of our research goals we formulated five hypotheses that have been validated in this dissertation. We briefly discuss whether we accept or reject the hypotheses:

- *Change type definition hypothesis* H1a: accepted
  We precisely defined change types according to the four basic tree edit operations in the abstract syntax tree: insert, delete, move, and update. Each change type received a meaningful name. In addition, the change type definitions were used to develop an algorithm that extracts source code changes in the abstract syntax tree by using tree differencing. Our emerged change distilling algorithm is based on the tree differencing algorithm of Chawathe *et al.* (1996).

- *Change type classification hypothesis* H1b: accepted
  According to the change type definition we classified each change type with one of the five change significance levels: none, low, medium, high, and crucial. The change significance level expresses the impact a change type may have on other source code entities and whether it may be functionality-preserving or functionality-modifying.

- *Co-change hypothesis* H2a: accepted
  Considering our experiments we gained insights of various commenting processes of software systems. Method body comments tend to co-change with source code in the same revision whereas re-documentation takes place for API comments.

- *Change activity hypothesis* H2b: partially accepted
  The results of our experiments showed that change type patterns can un-

cover different development activities. Because we realized that our approach is not robust enough against a strong mixture of different development activities, we partially accept this hypothesis.

- *Method invocation change hypothesis* H2c: accepted
  With our experiments we have shown that invocations of certain methods change often and are often involved in bug fixes. The proposed ranking highlighted such methods and we argued that the extracted method invocation change patterns can be leveraged to provide suggestive feedback for developers.

10.2 Achievement of Research Goals and Thesis

We summarize the justifications for the fulfillment of our research goals and the verification of our thesis.

Our taxonomy of source code changes precisely defines and classifies source code changes. It combines the results of Hypotheses H1a and H1b. We have therefore fulfilled Research Goal G1.

CHANGEDISTILLER, the implementation of our change distilling algorithm, extracts source code changes from the historical data of Java software systems. It is based on the tree differencing algorithm presented by Chawathe *et al.* (1996). We have adapted and improved the algorithm so that it can handle ASTs. To validate the change distilling algorithm, and CHANGEDISTILLER respectively, we developed a benchmark with 1,064 manually extracted change types. Compared to the original algorithm we improved the accuracy of the change extraction by 45 percent. We have therefore fulfilled Research Goal G2.

We have provided three software evolution experiments. We have shown that we gained interesting insights about the evolution of a software system in general and about the meaning of source code changes in particular. In addition, we have discussed how we can support software evolution by leveraging source code changes. We have therefore fulfilled Research Goal G3.

The three experiments have shown that the extraction of fine-grained source code changes by using tree differencing algorithm enabled us to contribute to the two dimensions of analyzing software evolution and to both perspectives of software engineering research. We therefore regard the thesis of this dissertation as verified.

10.3 Opportunities for Future Research

During the work on this dissertation we encountered promising further research directions:

- *Further programming languages*: Our CHANGEDISTILLER provides the extraction of source code changes for the Java programming languages. Therefore, all our experiments were conducted on software systems written in Java. To extend our empirical analyses we plan to use software systems written in different programming languages. We consider two ways to integrate new languages into CHANGEDISTILLER. First, we leverage different parsers to generate the intermediate ASTs necessary for change distilling. Using *gcc* will be a corresponding possibility.

  Second, we integrate CHANGEDISTILLER into other integrated development environments (IDE), such as, Visual Studio or EiffelStudio, to use their AST leveraging capabilities. A complete integration of CHANGEDISTILLER into an new IDE will require to re-implement it from scratch in the corresponding programming language. However, such an integration will be worth the effort because (1) IDEs guarantee a certain level of platform independence, (2) it is convenient to conduct studies by using IDEs, and (3) we are certain that it will open up interesting case studies.

- *Incremental change extraction*: So far we have extracted source code changes retroactively. That limits the possibility of analyzing source code changes on-the-fly. We consider to integrate the change extraction into the commit process. A promising platform to test this integration is the *Jazz* development platform.[1] *Jazz* comes with an integrated versioning and task management system. That means, it already provides a relational database that stores historical data. Integrating CHANGEDISTILLER into *Jazz* will enable the extraction of source code changes at commit time and support an on-the-fly analysis of the changes.

- *Empirical comparisons of change processes*: We plan to conduct empirical studies to reveal similarities of the change process of various software systems. For instance, we are eager to find out whether pre and post-release source code changes are related. Or, whether post-release failures can be traced back to quick pre-release changes.

- *Semantics of comment changes*: The co-change study of comments and source code is based on a quantitative exploration. Interesting questions exist that

[1] http://jazz.net

address the semantic of the comments: How does a source code change induce a comment change? Are complex methods or classes stronger commented than simple methods or classes and are such comments adapted faster? Does the amount of comments in a class impact the adaptation of them? Are invocations to complex or operation critic methods more frequently commented than others? Does the amount of comments in a superclass have any influences on the comments in its subclasses?

- *Change type pattern catalogue*: We plan to improve our approach to discover change type patterns: Since the used clustering approach is sensible to noise in the change type data we will apply filter mechanisms to separate outlier change types as well as try different clustering techniques. Furthermore, using a sliding time frame instead of yearly quarters shall improve the detection of local change type patterns. We expect that the improvements detect new patterns and extend our catalogue. A number of empirical studies with open-source and commercial software systems will also be conducted.

- *Recommendation systems*: The experiments presented in this dissertation provided a proof-of-concept that developing recommendation systems leveraging source code change data is worthwhile. The proposed tools will be implemented as prototypes in Eclipse. To validate their effectiveness we plan to conduct controlled user studies with undergraduate as well as graduate students and with developers from our industrial partners.

Appendices

Complete List of Change Types

A

We distinguish between change types in body-parts, *i.e.*, method body or class body, and declaration-parts, *i.e.*, method declaration, class declaration, or attribute declaration. The change significance level (change sign. lvl.) of several change types is split into *normal* and {*protected*, *public*} ({pro.,pub.}). Changes of source code entities defined as protected or public may have a stronger change impact on other entities than such defined as private. Change type names marked with * are functionality-preserving, all others functionality-modifying.

A.1 Body-Part Change Types

		Change sign. lvl.	
Change type	Operation	normal	{pro., pub.}
Additional Functionality*	$\langle$ INS$((l(M), n(M)), C, k)$, INS$((l(P(M)), \{\}), M, 4)$ $\rangle$	low	
Additional Object State*	$\langle$ INS$((l(A), n(A)), C, k)$ + INS$((l(T(A)), v(T(A))), A, 1)$ $\rangle$	low	
Loop Condition Expression Change	UPD$(l(L), v(CE_{new}(L)))$;	medium	
Control Structure Condition Expression Change	UPD$(l(CS), v(CE_{new}(CS)))$	medium	
Else-Part Insert	INS$((l(EP), v(CS)), CS, 1)$	medium	

Change type	Operation	Change sign. lvl. normal	{pro., pub.}
Else-Part Delete	DEL(EP)	medium	
Removed Functionality	DEL(M)	high	crucial
Removed Object State	DEL(A)	high	crucial
Statement Delete	DEL(s)	low	
Statement Insert	INS$((l(s), v(s)), y, k)$	low	
Statement Ordering Change	MOV$(s, p(s), k)$	low	
Statement Parent Change	MOV(s, y, k), $p_{old}(s) \neq p_{new}(s)$	medium	
Statement Update*	UPD(s, val)	low	
Comment Delete	DEL(CO)	none	
Comment Insert	INS$((l(CO), v(CO)), y, k)$	none	
Comment Move	MOV(CO, y, k), $p_{old}(CO) \neq p_{new}(CO)$	none	
Comment Update	UPD(CO, val)	none	

A.2 Declaration-Part Change Types

Change type	Operation	Change sign. lvl. normal	{pro., pub.}
Adding Attribute Modifiability	DEL$(\mu_F(A))$	low	
Adding Class Derivability	DEL$(\mu_F(C))$	low	
Adding Method Overridability	DEL$(\mu_F(M))$	low	
Class Renaming*	UPD$(C, v(n_{new}(C)))$	high	
Decreasing Accessibility	INS$((l(\mu_A), v(\mu_A)), y, 1)$; DEL$(\mu_A)$; UPD$(\mu_A, val)$	high	
Attribute Type Change	UPD$(T(A), v(T_{new}(A)))$	high	crucial
Attribute Renaming*	UPD$(n(A), v(n_{new}(A)))$	medium	high
Increasing Accessibility	INS$((l(\mu_A), v(\mu_A)), y, 1)$; DEL$(\mu_A)$; UPD$(\mu_A, val)$	medium	
Method Renaming*	UPD$(M, v(n_{new}(M)))$	medium	high

		Change sign. lvl.	
Change type	Operation	normal	{pro., pub.}
Parameter Delete	$\text{DEL}(\rho)$	high	crucial
Parameter Insert	$\langle\, \text{INS}((l(\rho), v(n(\rho))), P(M), k),$ $\text{INS}((l(T(\rho)), v(T(\rho))), \rho, 1)\,\rangle$	high	crucial
Parameter Ordering	$\text{MOV}(\rho, P(M), k)$	high	crucial
Change			
Parameter Type Change	$\text{UPD}(T(\rho), v(T_{new}(\rho)))$	crucial	
Parameter Renaming*	$\text{UPD}(\rho, v(n_{new}(\rho)))$	medium	
Parent Class Delete	$\text{DEL}(T), T \in p_{Cold}(C)$	high	crucial
Parent Class Insert	$\text{INS}((l(T), v(T)), p_C(C), k)$	crucial	
Parent Class Update	$\text{UPD}(T, v(T_{new})), T \in p_{Cold}(C)$	crucial	
Parent Interface Delete	$\text{DEL}(T), T \in p_{Iold}(C)$	crucial	
Parent Interface Insert	$\text{INS}((l(T), v(T)), p_I(C), k)$	crucial	
Parent Interface Update	$\text{UPD}(T, v(T_{new})), T \in p_{Iold}(C)$	crucial	
Removing Attribute Modifiability	$\text{INS}((l(\mu_F), v(\mu_F)), A, 2)$	high	crucial
Removing Class Derivability	$\text{INS}((l(\mu_F), v(\mu_F)), C, 2)$	crucial	
Removing Method Overridability	$\text{INS}((l(\mu_F), v(\mu_F)), M, 2)$	crucial	
Return Type Delete	$\text{DEL}(T(M))$	high	crucial
Return Type Insert	$\text{INS}((l(T(M)), v(T(M))), M, 3),$ $T_{old}(M) = \{\}$	high	crucial
Return Type Update	$\text{UPD}(T(M), v(T_{new}(M)))$	high	crucial
API Comment Delete	$\text{DEL}(AC(x)), x = \{C, M, A\}$		
API Comment Insert	$\text{INS}((l(AC), v(AC)), y, k),$ $y = \{C, M, A\}$	none	
API Comment Update	$\text{UPD}(AC(x), val), x = \{C, M, A\}$		

Selected Methods for the Benchmark

B

For the benchmark we selected source code changes in the history of eight methods from three different software systems. In this appendix we list the details of the systems and methods.

Software Systems

System	Period	# source revisions	# changes	LOC (release) first	LOC (release) last
ArgoUML	01/98∼12/05	39,421	183,752	200,735	239,791
Azureus	07/03∼05/07	33,008	245,214	17,227	362,316
Eclipse JDT	06/01∼05/07	40,303	371,713	201,477	361,771

Selected Methods

ArgoUML

- `org.argouml.uml.diagram.static_structure.ui.FigClass.getPopUp-Actions(MouseEvent)`, 32 revisions
- `org.argouml.persistence.ZargoFilePersister.loadProject(URL)`, 6 revisions
- `org.argouml.uml.reveng.java.Modeller.addOperation(short, String, String, Vector, String)`, 36 revisions

- `org.argouml.uml.ui.ActionOpenProject.actionPerformed(ActionEvent)`, 37 revisions

Azureus

- `org.gudy.azureus2.core3.download.impl.DownloadManagerImpl.readTorrent()`, 24 revisions
- `org.gudy.azureus2.core3.tracker.server.impl.TRTrackerServerTorrentImpl.TRTrackerServerPeerImpl(String, String, int, String, String, long, long, long, int, long)`, 31 revisions

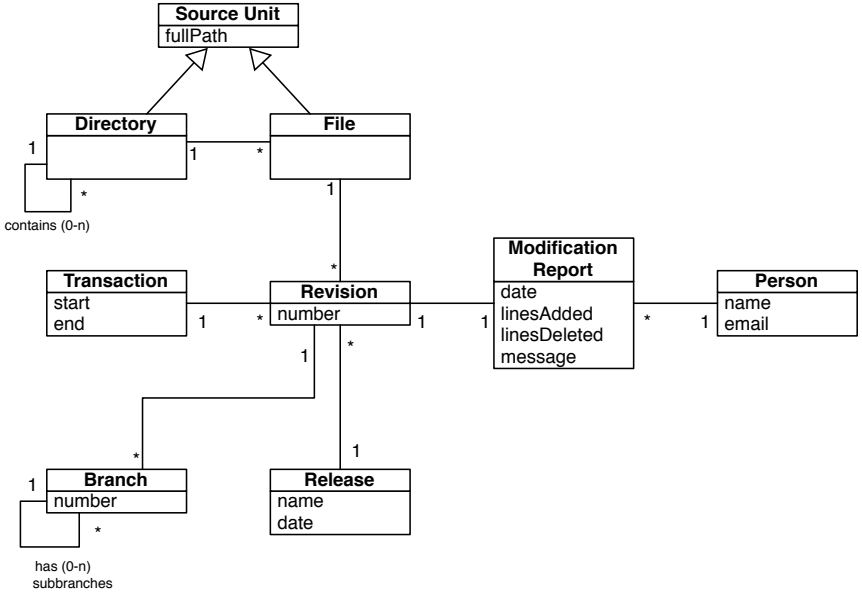
Eclipse JDT Core

- `org.eclipse.jdt.internal.compiler.ast.TypeDeclaration.resolve()`, 41 revisions
- `org.eclipse.jdt.internal.core.SelectionRequestor.acceptSourceMethod(IType, char[], char[][], char[][])`, 12 revisions

EVOLIZER Versioning Meta Model

C

The versioning meta model is part of the EVOLIZER platform. The corresponding package inside the platform is `org.evolizer.model.versioning`.



Comment to Code Mapping

D

For each system of Experiment 2 in Chapter 6, we list the contingency tables for the observed and expected values along with the χ^2 value. O = Observed, E = Expected, T = Total.

ArgoUML

O	Class	Field	Method	If	Loop	VD	Call	Other	Total
c	1,659	1,606	12,347	765	81	1,034	744	595	18,831
$\bar{c}$	33	1,852	0	9,469	1,531	10,577	17,255	42,832	83,414
T	1,692	3,458	12,212	10,234	1,612	11,611	17,999	43,427	102,245

E	Class	Field	Method	If	Loop	VD	Call	Other
c	312	637	2,249	1,885	297	2,138	3,315	7,998
$\bar{c}$	1,380	2,821	9,963	8,349	1,315	9,473	14,684	35,429

$\chi^2 = 76{,}269$

Azureus

O	Class	Field	Method	If	Loop	VD	Call	Other	Total
c	1,346	1,978	4,181	1,586	270	1,288	1,027	6,088	17,764
$\bar{c}$	1,314	7,899	19,789	16,439	3,256	19,208	31,299	92,119	191,323
T	2,660	9,877	23,970	18,025	3,526	20,496	32,326	98,207	209,087

E	Class	Field	Method	If	Loop	VD	Call	Other
c	226	839	2,036	1,531	300	1,741	2,746	8,344
$\bar{c}$	2,434	9,038	21,934	16,494	3,226	18,755	29,580	89,863

$\chi^2 = 12{,}200$

Eclipse Core

O	Class	Field	Method	If	Loop	VD	Call	Other	Total
c	708	1,412	5,456	982	153	788	602	2,558	12,659
$\bar{c}$	190	2,244	2,876	5,505	1,049	5,772	7,198	27,170	52,004
T	898	3,656	8,332	6,487	1,202	6,560	7,800	29,728	64,663

E	Class	Field	Method	If	Loop	VD	Call	Other
c	176	716	1,631	1,270	235	1,284	1,527	5,820
$\bar{c}$	722	2,940	6,701	5,217	967	5,276	6,273	23,908

$\chi^2 = 17{,}323$

Eclipse JDT

O	Class	Field	M	If	Loop	VD	Call	Other	Total
c	3,132	7,555	29,213	7,308	689	4,674	3,256	14,993	70,820
$\bar{c}$	2,648	25,172	28,301	77,066	11,173	67,706	81,460	264,520	558,046
T	5,780	32,727	57,514	84,374	11,862	72,380	84,716	279,513	628,866

E	Class	Field	Method	If	Loop	VD	Call	Other
c	651	3,686	6,477	9,502	1,336	8,151	9,540	31,477
$\bar{c}$	5,129	29,041	51,037	74,872	10,526	64,229	75,176	248,036

$\chi^2 = 122{,}162$, M = Method

Eclipse PDE

O	Class	Field	Method	If	Loop	VD	Call	Other	Total
c	467	722	4,912	1,202	174	1,042	1,219	3,149	12,887
$\bar{c}$	1,422	8,101	11,749	14,914	2,459	16,661	25,326	57,899	138,531
T	1,889	8,823	16,661	16,116	2,633	17,703	26,545	61,048	151,418

E	Class	Field	Method	If	Loop	VD	Call	Other
c	161	751	1,418	1,372	224	1,507	2,260	5,196
$\bar{c}$	1,728	8,072	15,243	14,744	2,409	16,196	24,286	55,852

$\chi^2 = 11{,}646$

jEdit

O	Class	Field	Method	If	Loop	VD	Call	Other	Total
c	600	1,158	5,631	825	86	538	410	2,013	11,261
$\bar{c}$	284	2,392	618	9,769	1,258	6,027	11,601	30,755	62,704
T	884	3,550	6,249	10,594	1,344	6,565	12,011	32,768	73,965

E	Class	Field	Method	If	Loop	VD	Call	Other
c	135	540	951	1,613	205	1000	1,829	4,989
$\bar{c}$	749	3,010	5,298	8,981	1,139	5,565	10,182	27,779

$\chi^2 = 34{,}060$

JFreeChart

O	Class	Field	Method	If	Loop	VD	Call	Other	Total
c	978	2,753	10,492	584	65	957	1,163	3,444	20,436
$\bar{c}$	0	51	0	7,567	702	13,912	20,680	40,951	83,863
T	978	2,804	10,492	8,151	767	14,869	21,843	44,395	104,299

E	Class	Field	Method	If	Loop	VD	Call	Other
c	192	549	2,056	1,597	150	2,913	4,280	8,699
$\bar{c}$	786	2,255	8,436	6,554	617	11,956	17,563	35,696

$\chi^2 = 67{,}325$

Webframework

O	Class	Field	Method	If	Loop	VD	Call	Other	Total
c	580	239	6,304	619	24	375	847	2,462	11,450
$\bar{c}$	368	2,352	2,799	3,956	714	8,406	26,594	28,320	73,509
T	948	2,591	9,103	4,575	738	8,781	27,441	30,782	84,959

E	Class	Field	Method	If	Loop	VD	Call	Other
c	128	349	1,227	617	99	1,183	3,698	4,149
$\bar{c}$	820	2,242	7,876	3,958	639	7,598	23,743	26,633

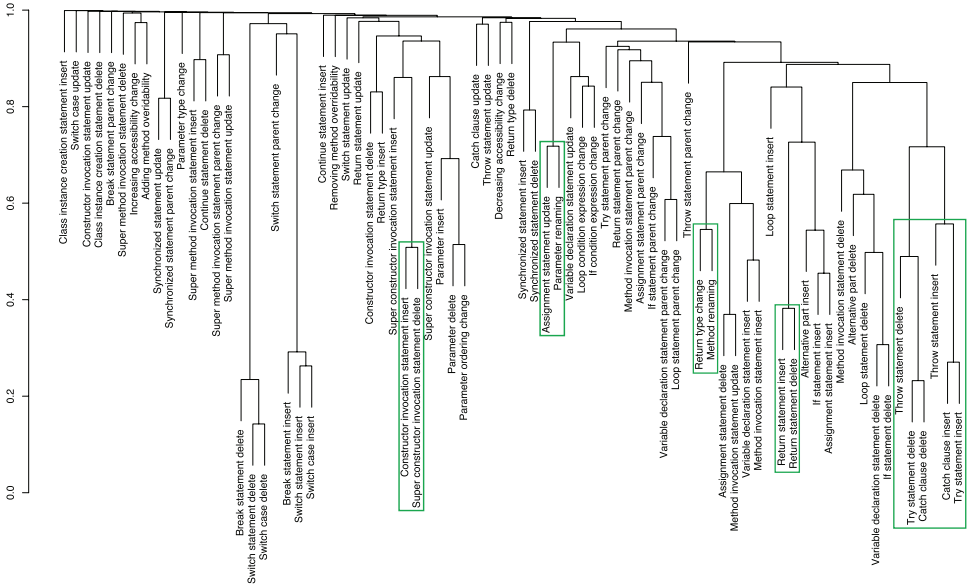
$\chi^2 = 30{,}213$

Change Type Clusters

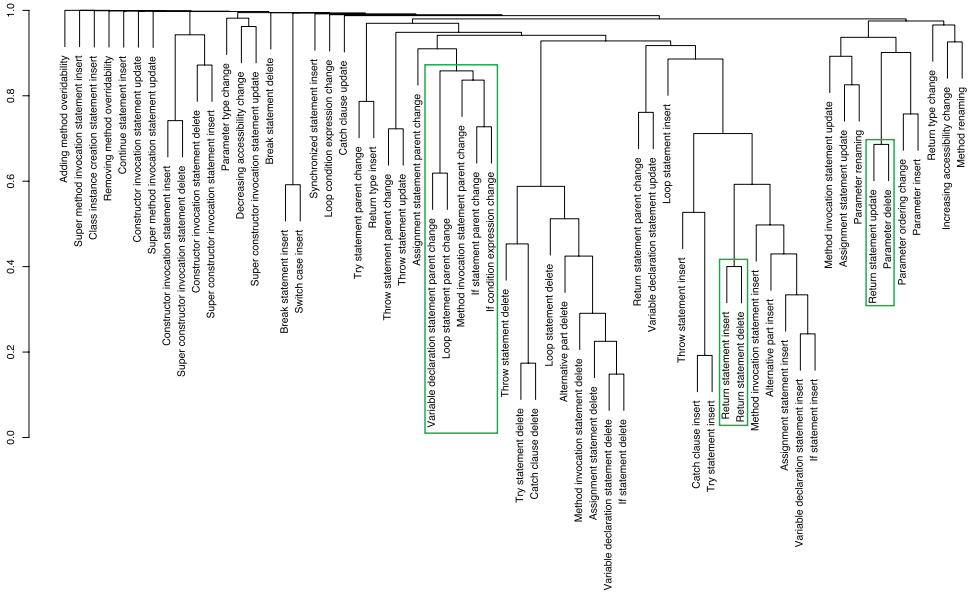
E

This appendix shows all dendrograms with highlighted change type patterns that we discussed in Chapter 7.

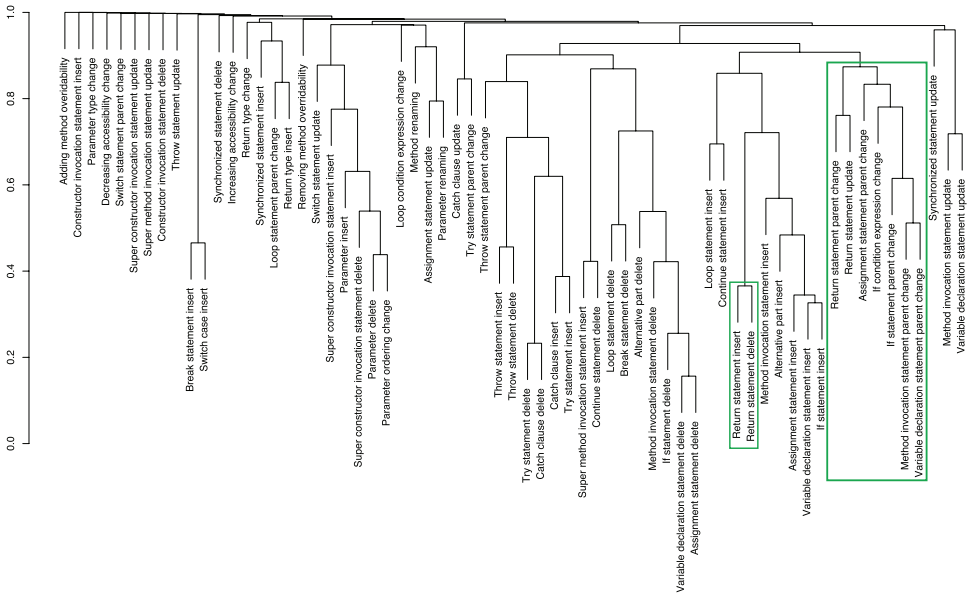
Webframework full Cluster



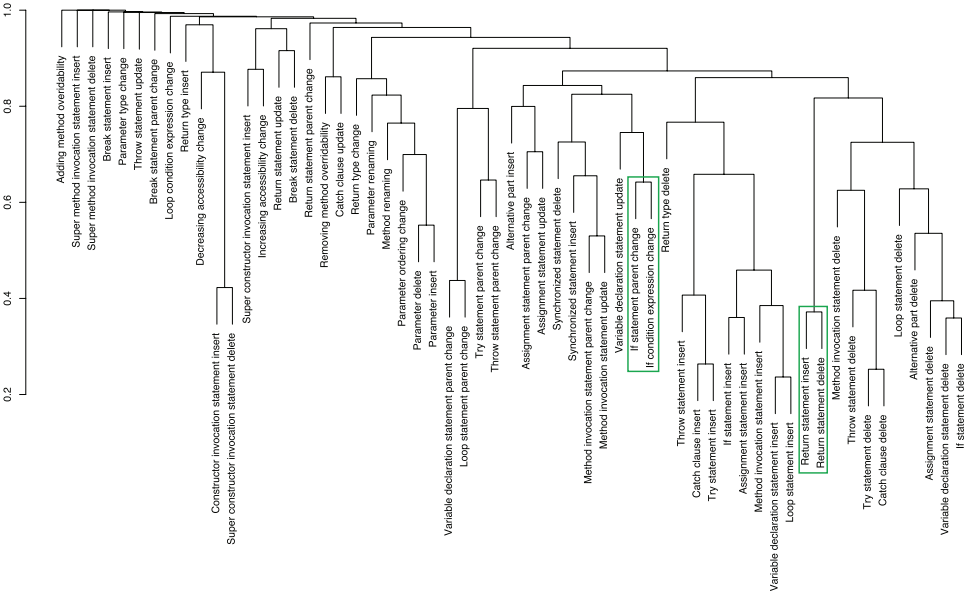
Webframework third quarter cluster of 2005



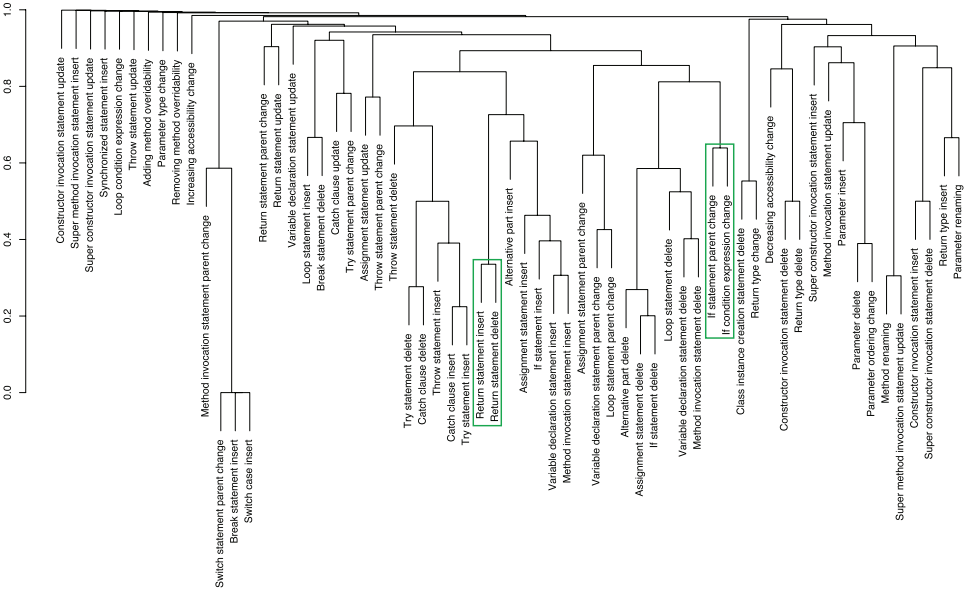
Webframework fourth quarter cluster of 2005



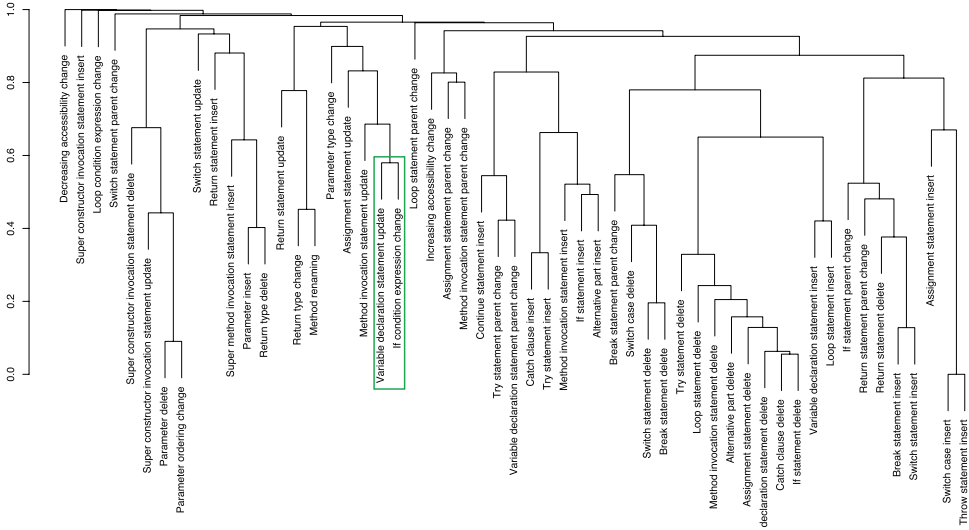
Webframework first quarter cluster of 2007



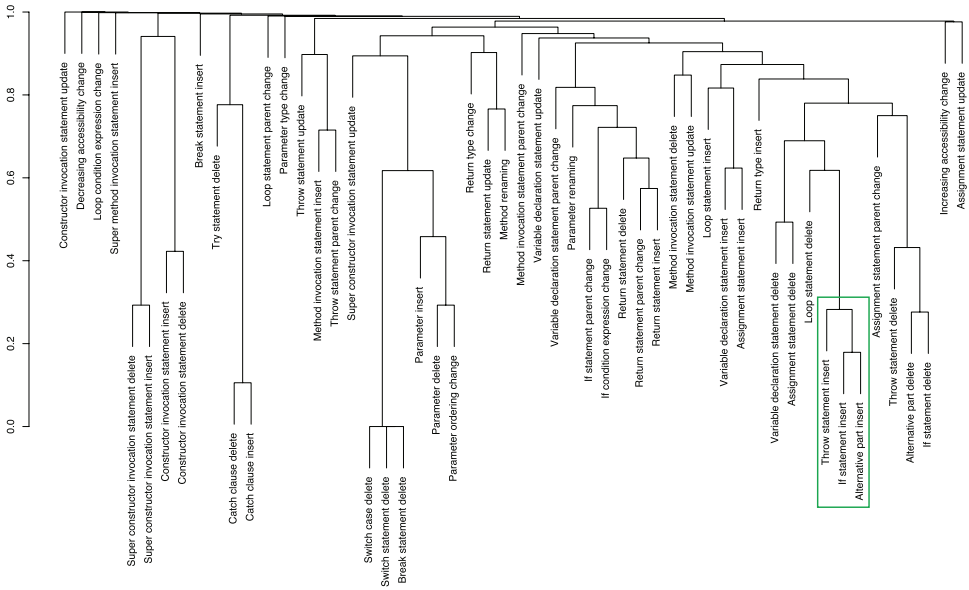
Webframework second quarter cluster of 2007



jEdit first quarter cluster of 2005



JFreeChart first quarter cluster of 2005



Method Ranking and Change Patterns

F

We list the results of Chapter 8 in more detail. S = Score determines the rank.

F.1 Method Ranking

Eclipse Core

Method	S	#context changes	#context fixes (%)	#update changes	#update fixes (%)
`Assert.isTrue(2)`	0.95	17	16 (94%)	93	20 (22%)
`StringBuffer.append(1)`	0.47	4	2 (50%)	66	27 (41%)
`<undef>.log(1)`	0.47	6	3 (50%)	69	24 (35%)
`Map.put(2)`	0.41	6	2 (33%)	55	31 (56%)
`Hashtable.put(2)`	0.29	20	20 (100%)	7	3 (43%)
`Assert.isTrue(1)`	0.27	1	1 (100%)	18	16 (89%)
`EP.internalError(1)`†	0.23	2	2 (100%)	24	4 (17%)
`PDR.parseProblem(1)`‡	0.22	1	1 (100%)	29	0 (0%)
`IPM.beginTask(2)`*	0.17	0	0 (0%)	134	23 (17%)
`ArrayList.add(1)`	0.16	5	1 (20%)	28	13 (46%)

†`EP` = `ExtensionsParser`, ‡`PDR` = `ProjectDescriptionReader`
*`IPM` = `IProgressMonitor`

Eclipde JDT

Method	S	#context changes	#context fixes (%)	#update changes	#update fixes (%)
`StringBuffer.append(1)`	0.74	581	230 (40%)	1958	669 (34%)
`<undef>.log(1)`	0.26	37	35 (95%)	351	273 (78%)
`List.add(1)`	0.21	178	82 (46%)	510	165 (32%)
`Map.put(2)`	0.21	98	34 (35%)	612	246 (40%)
`IPM.beginTask(2)`†	0.16	13	6 (46%)	623	117 (19%)
`System.arraycopy(5)`	0.15	124	53 (43%)	506	98 (19%)
`Button.setText(1)`	0.15	11	7 (64%)	428	99 (23%)
`PR.handle(5)`‡	0.14	16	11 (69%)	371	84 (23%)
`VPS.println(2)`*	0.13	0	0 (0%)	333	332 (100%)
`Button.addSelLis(1)`¶	0.12	13	9 (69%)	295	98 (33%)

†`IPM` = `IProgressMonitor`, ‡`PR` = `ProblemReporter`
*`VPS` = `VerbosePacketStream`, ¶`addSelLis` = `addSelectionListener`

Eclipse PDE

Method	S	# context changes	# context fixes (%)	# update changes	# update fixes (%)
`PrintWriter.println(1)`	0.42	41	10 (24%)	490	87 (18%)
`StringBuffer.append(1)`	0.30	78	27 (35%)	215	43 (20%)
`ArrayList.add(1)`	0.26	57	18 (32%)	204	45 (22%)
`Map.put(2)`	0.23	14	7 (50%)	136	44 (32%)
`Label.setText(1)`	0.18	7	3 (43%)	195	11 (6%)
`Button.addSelLis(1)`†	0.15	9	2 (22%)	199	34 (17%)
`List.add(1)`	0.15	15	7 (47%)	64	34 (53%)
`<undef>.showWhile(2)`	0.14	2	1 (50%)	106	20 (19%)
`BER.report(3)`‡	0.13	9	9 (100%)	43	16 (37%)
`Button.setText(1)`	0.11	4	1 (25%)	207	8 (4%)

†`addSelLis` = `addSelectionListener`, ‡`BER` = `BundelErrorReporter`

F.2 Context Change Patterns

Eclipse Core

		CC.Qual		CC.Args		CC.Both		CC.Other	
Method	#	%n	%f	%n	%f	%n	%f	%n	%f
`Assert.isTrue(2)`	17	0	0	0	0	0	0	6	94
`StringBuffer.append(1)`	4	0	0	25	0	0	0	25	50
`<undef>.log(1)`	6	0	0	0	0	0	0	63	38
`Map.put(2)`	6	17	0	17	17	0	0	33	17
`Hashtable.put(2)`	20	0	0	0	25	0	0	0	75
`Assert.isTrue(1)`	1	0	0	0	0	0	0	0	100
`EP.internalError(1)`†	2	0	0	0	0	0	0	0	100
`PDR.parseProblem(1)`‡	1	0	0	0	0	0	0	0	100
`IPM.beginTask(2)`*	0	0	0	0	0	0	0	0	0
`ArrayList.add(1)`	5	13	0	0	0	0	0	75	13

†`EP` = `ExtensionsParser`, ‡`PDR` = `ProjectDescriptionReader`
*`IPM` = `IProgressMonitor`

Eclipse JDT

		CC.Qual		CC.Args		CC.Both		CC.Other	
Method	#	%n	%f	%n	%f	%n	%f	%n	%f
`String.append(1)`	581	0	0	4	3	0	0	55	37
`<undef>.log(1)`	37	0	5	5	62	0	7	0	21
`List.add(1)`	178	0	1	13	12	3	3	39	28
`Map.put(2)`	98	2	2	24	9	6	1	35	21
`IPM.beginTask(2)`†	13	20	7	0	0	0	0	27	47
`System.arraycopy(5)`	124	0	0	32	34	0	0	26	8
`Button.setText(1)`	11	0	0	0	0	0	0	53	47
`PR.handle(5)`‡	16	10	50	0	0	0	0	15	25
`VPS.println(2)`*	0	0	0	0	0	0	0	0	0
`Button.addSelLis(1)`¶	13	0	0	0	0	0	0	43	57

†`IPM` = `IProgressMonitor`, ‡`PR` = `ProblemReporter`
*`VPS` = `VerbosePacketStream`, ¶`addSelLis` = `addSelectionListener`

Eclipse PDE

		CC.Qual		CC.Args		CC.Both		CC.Other	
Method	#	%n	%f	%n	%f	%n	%f	%n	%f
`PrintWriter.println(1)`	41	2	0	0	0	0	0	73	25
`StringBuffer.append(1)`	78	0	2	7	5	0	0	63	24
`ArrayList.add(1)`	57	1	0	9	10	6	3	49	22
`Map.put(2)`	14	0	0	14	7	7	0	29	43
`Label.setText(1)`	7	0	0	0	0	0	0	44	56
`Button.addSelLis(1)`†	9	0	0	0	0	0	0	78	22
`List.add(1)`	15	0	0	6	0	6	0	35	53
`<undef>.showWhile(2)`	2	0	0	0	0	0	0	50	50
`BER.report(3)`‡	9	0	0	0	0	0	0	0	100
`Button.setText(1)`	4	0	0	0	0	0	0	75	25

†`addSelLis` = `addSelectionListener`, ‡`BER` = `BundelErrorReporter`

F.3 Update Change Patterns

Eclipse Core

		UC.Name		UC.Args		UC.Both	
Method	#	%n	%f	%n	%f	%n	%f
`Assert.isTrue(2)`	93	0	0	78	22	0	0
`StringBuffer.append(1)`	66	0	0	59	41	0	0
`<undef>.log(1)`	69	0	0	65	35	0	0
`Map.put(2)`	55	0	0	44	55	0	2
`Hashtable.put(2)`	7	0	14	57	29	0	0
`Assert.isTrue(1)`	18	0	0	11	89	0	0
`EP.internalError(1)`†	24	0	0	83	0	0	17
`PDR.parseProblem(1)`‡	29	0	0	100	0	0	0
`IPM.beginTask(2)`*	134	0	0	82	17	1	0
`ArrayList.add(1)`	28	0	0	54	46	0	0

†`EP` = `ExtensionsParser`, ‡`PDR` = `ProjectDescriptionReader`

*`IPM` = `IProgressMonitor`

Eclipse JDT

		UC.Name		UC.Args		UC.Both	
Method	#	%n	%f	%n	%f	%n	%f
`String.append(1)`	1958	0	0	66	34	0	0
`<undef>.log(1)`	351	5	32	17	46	0	0
`List.add(1)`	510	0	1	67	30	1	2
`Map.put(2)`	612	0	0	59	40	0	0
`IPM.beginTask(2)`†	608	0	0	80	17	1	1
`System.arraycopy(5)`	506	0	0	81	19	0	0
`Button.setText(1)`	428	1	0	75	23	0	0
`PR.handle(5)`‡	371	0	0	77	23	0	0
`VPS.println(2)`*	333	0	0	0	100	0	0
`Button.addSelLis(1)`¶	291	0	0	67	31	0	1

†`IPM` = `IProgressMonitor`, ‡`PR` = `ProblemReporter`
*`VPS` = `VerbosePacketStream`, ¶`addSelLis` = `addSelectionListener`

Eclipse PDE

		UC.Name		UC.Args		UC.Both	
Method	#	%n	%f	%n	%f	%n	%f
`PrintWriter.println(1)`	490	0	0	80	17	2	1
`StringBuffer.append(1)`	215	0	0	80	20	0	0
`ArrayList.add(1)`	204	0	0	72	21	6	1
`Map.put(2)`	136	0	0	68	32	0	0
`Label.setText(1)`	195	0	0	94	6	0	0
`Button.addSelLis(1)`†	199	0	0	83	17	0	0
`List.add(1)`	64	0	0	45	53	2	0
`<undef>.showWhile(2)`	106	0	0	81	19	0	0
`BER.report(3)`‡	43	0	0	63	37	0	0
`Button.setText(1)`	207	0	0	96	4	0	0

†`addSelLis` = `addSelectionListener`, ‡`BER` = `BundelErrorReporter`

G Publications

This appendix present the list of publications on which this dissertation is based on.

G.1 Journal Article

Change Distilling

(Fluri *et al.*, 2007a) Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, **33**(11), 725–743, November 2007.

G.2 Conference Papers

Change Distilling

(Fluri and Gall, 2006) Beat Fluri and Harald C. Gall. Classifying Change Types for Qualifying Change Couplings. In *Proceedings of the 9th International Conference on Program Comprehension*, pages 35–45. IEEE Computer Society, June 2006.

Co-Change (Fluri *et al.*, 2007b) Beat Fluri, Michael Würsch, and Harald C. Gall. Do Code and Comments Co-Evolve? On the Relation Between Source Code and Comment Changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 70–79. IEEE Computer Society, November 2007.

Change Type Patterns (Fluri *et al.*, 2008b) Beat Fluri, Emanuel Giger, and Harald C. Gall. Discovering Patterns of Change Types. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, short paper, to appear. IEEE Computer Society, September 2008.

G.3 Workshop Papers

Structural Change (Fluri *et al.*, 2005) Beat Fluri, Harald C. Gall, and Martin Pinzger. Fine-Grained Analysis of Change Couplings. In *Proceedings of the 5th International Workshop on Source Code Analysis and Manipulation*, pages 66–74. IEEE Computer Society, October 2005.

G.4 Technical Reports

Co-Change (Fluri *et al.*, 2008a) Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C. Gall. Analyzing the Co-Evolution of Comments and Source Code. Technical report, University of Zurich, May 2008.

Change-Affected Method Invocations (Fluri and Gall, 2008) Beat Fluri and Harald C. Gall. How Can We Benefit from Change Histories to Facilitate the Use of Methods? Technical report, University of Zurich, April 2008.

Bibliography

Adamson, G. W. and Boreham, J. (1974). The use of an association measure based on character structure to identify semantically related pairs of words and document titles. *Information Storage and Retrieval*, **10**(7-8), 253–260.

Anquetil, N. and Lethbridge, T. C. (1998). Extracting concepts from file names: A new file clustering criterion. In *Proceedings of the 20th International Conference on Software Engineering*, pages 84–93. IEEE Computer Society.

Anquetil, N. and Lethbridge, T. C. (1999). Experiments with clustering as a software remodularization method. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 235–255. IEEE Computer Society.

Antoniol, G., Canfora, G., Casazza, G., Lucia, A. D., and Merlo, E. M. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, **28**(10), 970–983.

Apiwattanapong, T., Orso, A., and Harrold, M. J. (2007). JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, **14**(1), 3–36.

Asklund, U. (1994). Identifying conflicts during structural merge. In *Proceedings of the Nordic Workshop on Programming Environment Research*, pages 231–242.

Baresi, L. and Morasca, S. (2007). Three empirical studies on estimating the design effort of Web applications. *ACM Transactions on Software Engineering and Methodology*, **16**(4), 15.

Baxter, I. D., Yahin, A., de Moura, L. M., Sant'Anna, M., and Bier, L. (1998). Clone detection using abstract syntax trees. In *Proceedings of the 14th International Conference on Software Maintenance*, pages 368–377. IEEE Computer Society.

Bernstein, A., Kiefer, C., and Kaufmann, E. (2005). SimPack: A generic Java library for similarity measures in ontologies. Technical report, University of Zurich, Switzerland.

Bevan, J., E. James Whitehead, J., Kim, S., and Godfrey, M. W. (2005). Facilitating software evolution research with Kenyon. In *Proceedings of the Joint 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 177–186. ACM.

Bille, P. (2005). A survey on tree edit distance and related problems. *Theoretical Computer Science*, **337**(1–3), 217–239.

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice Hall.

Boehm, B. W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, **21**(5), 61–72.

Breu, S. and Zimmermann, T. (2006). Mining aspects from version history. In *Proceedings of the 21st International Conference on Automated Software Engineering*, pages 221–230. IEEE Computer Society.

Brooks, F. P. (1995). *The Mythical Man-Month*. Addison Wesley Longman, Inc., anniversary edition.

Canfora, G., Cerulo, L., and Penta, M. D. (2007). Identifying changed source code lines from version repositories. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, page 14. IEEE Computer Society.

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504. ACM.

Dagenais, B. and Robillard, M. P. (2008). Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering*, page to appear. ACM.

Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. (2004). Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th Annual Symposium on Computational Geometry*, pages 253–262. ACM.

Davis, A. M. (1995). *201 Principles of Software Development*. McGraw-Hill.

DeLine, R., Khella, A., Czerwinski, M., and Robertson, G. (2005). Towards understanding programs through wear-based filtering. In *Proceedings of the ACM Symposium on Software Visualization*, pages 183–192. ACM.

Demeyer, S., Tichelaar, S., and Ducasse, S. (2001). FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern.

Demeyer, S., Ducasse, S., and Nierstraz, O. (2003). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann.

des Rivières, J. and Wiegand, J. (2004). Eclipse: A platform for integrating development tools. *IBM Systems Journal*, **43**(2), 371–383.

Dice, L. R. (1945). Measures of the amount of ecologic association between species. *ESA Ecology*, **26**(3), 297–302.

Dig, D., Comertoglu, C., Marinov, D., and Johnson, R. (2006). Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 404–428. Springer.

Dowdy, S., Wearden, S., and Chilko, D. (2004). *Statistics for Research*. John Wiley & Sons, Inc.

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A. (2001). Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, **27**(1), 1–12.

Elshoff, J. L. and Marcotty, M. (1982). Improving computer program readability to aid modification. *Communications of the ACM*, **25**(8), 512–521.

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, **2**(3), 17–23.

Fischer, M. (2006). *EvoZilla—Longitudinal Evolution Analysis of Large Scale Software Systems*. Ph.D. thesis, Vienna University of Technology.

Fischer, M. and Gall, H. (2004). Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, **16**(6), 385–403.

Fischer, M., Pinzger, M., and Gall, H. (2003a). Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 90–99. IEEE Computer Society.

Fischer, M., Pinzger, M., and Gall, H. (2003b). Populating a release history database from version control and bug tracking systems. In *Proceedings of the 19th International Conference on Software Maintenance*, pages 23–32. IEEE Computer Society.

Fluri, B. and Gall, H. C. (2006). Classifying change types for qualifying change couplings. In *Proceedings of the 9th International Conference on Program Comprehension*, pages 35–45. IEEE Computer Society.

Fluri, B. and Gall, H. C. (2008). How can we benefit from change histories to facilitate the use of methods? Technical report, University of Zurich.

Fluri, B., Gall, H. C., and Pinzger, M. (2005). Fine-grained analysis of change couplings. In *Proceedings of the 5th International Workshop on Source Code Analysis and Manipulation*, pages 66–74. IEEE Computer Society.

Fluri, B., Würsch, M., Pinzger, M., and Gall, H. C. (2007a). Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, **33**(11), 725–743.

Fluri, B., Würsch, M., and Gall, H. C. (2007b). Do code and comments co-evolve? On the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 70–79. IEEE Computer Society.

Fluri, B., Würsch, M., Giger, E., and Gall, H. C. (2008a). Analyzing the co-evolution of comments and source code. Technical report, University of Zurich.

Fluri, B., Giger, E., and Gall, H. C. (2008b). Discovering patterns of change types. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, page to appear (short paper). IEEE Computer Society.

Gall, H., Hayek, K., and Jazayeri, M. (1998). Detection of logical coupling based on product release history. In *Proceedings of the 14th International Conference on Software Maintenance*, pages 190–198. IEEE Computer Society.

Gionis, A., Indyk, P., and Motwani, R. (1999). Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases*, pages 518–529. Morgan Kaufmann.

Gîrba, T. (2005). *Modeling History to Understand Software Evolution*. Ph.D. thesis, University of Bern.

Gîrba, T. and Ducasse, S. (2006). Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, **18**(3), 207–236.

Gîrba, T., Kuhn, A., Seeberger, M., and Ducasse, S. (2005). How developers drive software evolution. In *Proceedings of the 8th International Workshop on Principles of Software Evolution*, pages 113–122. IEEE Computer Society.

Godfrey, M. W. and Zou, L. (2005). Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, **31**(2), 166–181.

Goldberg, A. (1987). Programmer as reader. *IEEE Software*, **4**(5), 62–70.

Halstead, M. H. (1977). *Elements of Software Science*. Elsevier Science, Inc.

Hassan, A. E. and Holt, R. C. (2004). Studying the evolution of software systems using evolutionary code extractors. In *Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 76–81. IEEE Computer Society.

Holmes, R., Walker, R. J., and Murphy, G. C. (2006). Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, **32**(12), 952–970.

Horwitz, S. (1990). Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245. ACM.

Horwitz, S. B., Reps, T., and Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, **12**(1), 35–46.

Hovemeyer, D. and Pugh, W. (2004). Finding bugs is easy. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Progamming, Systems, Languages, and Applications*, pages 92–106. ACM.

Hunt, J. J. (2001). *Extensible, Language-Aware Differencing and Merging*. Ph.D. thesis, University of Karlsruhe.

Hunt, J. W. and McIlroy, M. D. (1976). An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories.

Hunt, J. W. and Szymanski, T. G. (1977). A fast algorithm for computing longest common subsequences. *Communications of the ACM*, **20**(5), 350–353.

Hutchens, D. H. and Basili, V. R. (1985). System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, **11**(8), 749–757.

Jaccard, P. (1912). The distribution of the flora in the alpine zone. *New Phytologist*, **11**(2), 37–50.

Jackson, D. and Ladd, D. A. (1994). Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the 10th International Conference on Software Maintenance*, pages 243–252. IEEE Computer Society.

Jain, A. K., Murty, M. N., and Flynn, P. J. (1999). Data clustering: A review. *ACM Computing Surveys*, **31**(3), 264–323.

Jakob, M. (2007). *Investigating Change Patterns that Fix Bugs*. Master's thesis, University of Zurich.

Jiang, Z. M. and Hassan, A. E. (2006). Examining the evolution of code comments in PostgreSQL. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, pages 179–180. ACM.

Kaelbling, M. J. (1988). Programming languages should NOT have comment statements. *ACM SIGPlan Notices*, **23**(10), 59–60.

Kagdi, H., Collard, M. L., and Maletic, J. I. (2005). Towards a taxonomy of approaches for mining of source code repositories. In *Proceedings of the 2nd International Workshop on Mining Software Repositories*, pages 1–5. ACM.

Kagdi, H., Collard, M. L., and Maletic, J. I. (2007). An approach to mining call-usage patterns with syntactic context. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, pages 457–460. ACM.

Kelter, U., Wehren, J., and Niere, J. (2005). A generic difference algorithm for UML models. In *Software Engineering*, pages 105–116.

Kim, M. and Notkin, D. (2006). Program element matching for multi-version program analyses. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, pages 58–64. ACM.

Kim, M., Notkin, D., and Grossman, D. (2007a). Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th International Conference on Software Engineering*, pages 333–343. IEEE Computer Society.

Kim, M., Beall, J., and Notkin, D. (2007b). Discovering and representing logical structure in code change. Technical report, University of Washington.

Kim, S. and Ernst, M. D. (2007). Which warnings should I fix first? In *Proceedings of the Joint 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 45–54. ACM.

Kim, S., Pan, K., and E. James Whitehead, J. (2005). When functions change their names: Automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152. IEEE Computer Society.

Kim, S., Zimmermann, T., Pan, K., and Whitehead, E. J. (2006a). Automatic identification of bug introducing changes. In *Proceedings of the 21th International Conference on Automated Software Engineering*, pages 81–90. IEEE Computer Society.

Kim, S., Pan, K., and Whitehead, E. J. (2006b). Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 35–45. ACM.

Kim, S., Whitehead, E. J., and Zhang, Y. (2008). Classifying software changes: Clean or buggy. *IEEE Transactions on Software Engineering*, **34**(2), 181–196.

Koschke, R. and Eisenbarth, T. (2000). A framework for experimental evaluation of clustering techniques. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 201–210. IEEE Computer Society.

Kuhn, A., Ducasse, S., and Girba, T. (2007). Semantic clustering: Identifying topics in source code. *Information and Software Technology*, **49**(3), 230–243.

Lakhotia, A. (1993). Understanding someone else's code: Analysis and experience. *Journal of Systems and Software*, **23**(3), 269–275.

Lawrie, D. J., Feild, H., and Binkley, D. (2006). Leveraged quality assessment using information retrieval techniques. In *Proceedings of the International Conference on Program Comprehension*, pages 149–158. IEEE Computer Society.

Lehman, M. M. (1980). Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, **68**(9), 1060–1076.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, **10**, 707–710.

Livshits, B. and Zimmermann, T. (2005). DynaMine: Finding common error patterns by mining software revision histories. In *Proceedings of the Joint 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 296–305. ACM.

Madhavji, N. H., Fernandez-Ramil, J., and Perry, D. E., editors (2006). *Software Evolution and Feedback: Theory and Practice*. John Wiley & Sons, Inc.

Maletic, J. I. and Collard, M. L. (2004). Supporting source code difference analysis. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 210 – 219. IEEE Computer Society.

Maletic, J. I. and Marcus, A. (2001). Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112. IEEE Computer Society.

Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y.-F., and Gansner, E. R. (1998). Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 45–54. IEEE Computer Society.

Maqbool, O. and Bari, H. A. (2007). Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, **33**(11), 759–780.

Marcus, A. and Maletic, J. I. (2001). Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering*, pages 107–114. IEEE Computer Society.

Marcus, A. and Maletic, J. I. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 125–135. IEEE Computer Society.

Marcus, A. and Poshyvanyk, D. (2005). The conceptual cohesion of classes. In *Proceedings of the 21st International Conference on Software Maintenance*, pages 133–142. IEEE Computer Society.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, **2**(4), 308–320.

Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, **28**(5), 449–462.

Mens, T. and Demeyer, S., editors (2008). *Software Evolution*. Springer.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition.

Müller, H. A. and Klashinsky, K. (1988). Rigi – A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86. IEEE Computer Society.

Naur, P. and Randall, B., editors (1968). *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*.

Neamtiu, I., Foster, J. S., and Hicks, M. (2005). Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2nd International Workshop on Mining Software Repositories*, pages 1–5. ACM.

Osterweil, L. J. (2007). A future of software engineering? In *Proceedings of Future of Software Engineering*, pages 1–11. IEEE Computer Society.

Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279–287. IEEE Computer Society.

Pfleeger, S. L. and Atlee, J. M. (2006). *Software Engineering – Theory and Practice*. Pearson Education, Inc., 3rd edition.

Phattarsukol, S. and Muenchaisri, P. (2001). Identifying candidate objects using hierarchical clustering analysis. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference*, pages 381–389. IEEE Computer Society.

Pinzger, M., Gall, H. C., and Fischer, M. (2005a). Towards an integrated view on architecture and its evolution. *Electronic Notes in Theoretical Computer Science*, **127**(3), 183–196.

Pinzger, M., Gall, H., Fischer, M., and Lanza, M. (2005b). Visualizing multiple evolution metrics. In *Proceedings of the ACM Symposium on Software Visualization*, pages 67–75. ACM.

Pinzger, M., Giger, E., and Gall, H. C. (2007). Handling unresolved method bindings in Eclipse. Technical report, University of Zurich.

Purushothaman, R. and Perry, D. E. (2005). Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, **31**(6), 511–526.

Raghavan, S., Rohana, R., Leon, D., Podgurski, A., and Augustine, V. (2004). Dex: A semantic-graph differencing tool for studying changes in large code base. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 188–197. IEEE Computer Society.

Ren, X. and Ryder, B. G. (2007). Heuristic ranking of Java program edits for fault localization. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 239–249. ACM.

Ren, X., Sha, F., Tip, F., Ryder, B. G., and Chesley, O. (2004). Chianti: A tool for change impact analysis of Java programs. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Progamming, Systems, Languages, and Applications*, pages 432–448. ACM.

Robbes, R. and Lanza, M. (2007a). A change-based approach to software evolution. *Eletronic Notes in Theoretical Computerscience*, **166**(1), 93–106.

Robbes, R. and Lanza, M. (2007b). Characterizing and understanding development sessions. In *Proceedings of the 15th International Conference on Program Comprehension*, pages 155–166. IEEE Computer Society.

Robbes, R. and Lanza, M. (2008). SpyWare: A change-aware development toolset. In *Proceedings of the 30th International Conference on Software Engineering*, pages 847–850. ACM.

Robillard, M. P. and Murphy, G. C. (2007). Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, **16**(1), 3.

Royce, W. W. (1970). Managing the development of large software systems: Concepts and techinques. In *Proceedings of the IEEE WESTCON*. IEEE Computer Society.

Sager, T., Bernstein, A., Pinzger, M., and Kiefer, C. (2006). Detecting similar Java classes using tree algorithms. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, pages 65–71. IEEE Computer Society.

Schreck, D., Dallmeier, V., and Zimmermann, T. (2007). How documentation evolves over time. In *Proceedings of the 9th International Workshop on Principles of Software Evolution*, pages 4–10. ACM.

Schwanke, R. W. and Hanson, S. J. (1994). using neural networks to modularize software. *Machine Learning*, **15**(2), 137–168.

Shasha, D. and Zhang, K. (1990). Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, **11**(4), 581–621.

Siff, M. and Reps, T. (1999). Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, **25**(6), 749–768.

Śliwerski, J., Zimmermann, T., and Zeller, A. (2005a). Hatari: Raising risk awareness. In *Proceedings of the Joint 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 107–110. ACM.

Śliwerski, J., Zimmermann, T., and Zeller, A. (2005b). When do changes induce fixes? In *Proceedings of the 2nd International Workshop on Mining Software Repositories*, pages 24–28. ACM.

Spinellis, D. (2006). *Code Quality—The Open Source Perspective*. Addison-Wesley, Pearson Education, Inc.

Stoerzer, M., Ryder, B. G., Ren, X., and Tip, F. (2006). Finding failure-inducing changes in Java programs using change classification. In *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 57–68. ACM.

Tai, K.-C. (1979). The tree-to-tree correction problem. *Journal of the Association for Computing Machinery*, **26**(3), 422–433.

Tan, L., Yuan, D., Krishna, G., and Zhou, Y. (2007). /* iComment: Bugs or bad comments? */. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 145–158. ACM.

Tenny, T. (1988). Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering*, **14**(9), 1271–1279.

Tu, Q. and Godfrey, M. W. (2002). An integrated approach for studying architectural evolution. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 127–136. IEEE Computer Society.

Valiente, G. (2002). *Algorithms on Trees and Graphs*. Springer.

van Deursen, A. and Kuipers, T. (1999). Identifying objects using cluster and concept analysis. In *Proceedings of the 21st International Conference on Software Engineering*, pages 246–255. ACM.

Vanter, M. L. V. D. (2002). The documentary structure of source code. *Information and Software Technology*, **44**(13), 767–782.

Čubranić, D., Murphy, G. C., Singer, J., and Booth, K. S. (2005). Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, **31**(6), 446–465.

Wasylkowski, A., Zeller, A., and Lindig, C. (2007). Detecting object usage anomalies. In *Proceedings of the Joint 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 35–44. ACM.

Weidl, J. and Gall, H. C. (1998). Binding object models to source code: An approach to object-oriented re-architecting. In *Proceedings of the 22nd International Computer Software and Applications Conference*, pages 26–31. IEEE Computer Society.

Weissgerber, P. and Diehl, S. (2006). Identifying refactorings from source-code changes. In *Proceedings of the 21st International Conference on Automated Software Engineering*, pages 231–240. IEEE Computer Society.

Westfechtel, B. (1991). Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd International Workshop Software Configuration Management*, pages 68–79. ACM.

Wille, R. (1981). Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered Sets*, pages 445–470. Dordrecht-Boston.

Williams, C. C. and Hollingsworth, J. K. (2005). Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, **31**(6), 466–480.

Witte, R., Zhang, Y., and Rilling, J. (2007). Empowering software maintainers with semantic web technologies. In *Proceedings of the 4th European Semantic Web Conference*, pages 37–52. Springer.

Würsch, M. (2006). *Improving ChangeDistiller—Improving Abstract Syntax Tree based Source Code Change Detection*. Master's thesis, University of Zurich.

Xing, Z. and Stoulia, E. (2005a). Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering*, **31**(10), 850–868.

Xing, Z. and Stoulia, E. (2005b). UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th International Conference on Automated Software Engineering*, pages 54–65. ACM.

Xu, X., Lung, C.-H., Zaman, M., and Srinivasan, A. (2004). Program restructuring through clustering techniques. In *Proceedings of the 4th International Workshop on Source Code Analysis and Manipulation*, pages 75–84. IEEE Computer Society.

Yang, W. (1991). Identifying syntactic differences between two programs. *Software–Practice and Experience*, **21**(7), 739–755.

Yang, W. (1994). How to merge program texts. *Journal of Systems and Software*, **27**(2), 129–135.

Ying, A. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C. (2004). Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, **30**(9), 574–586.

Ying, A. T. T., Wright, J. L., and Abrams, S. (2005). Source code that talks: An exploration of eclipse task comments and their implication to repository mining. In *Proceedings of the 2nd International Workshop on Mining Software Repositories*, pages 1–5.

Zeller, A. (2007). The future of programming environments: Integration, synergy, and assistance. In *Proceedings of Future of Software Engieering*, pages 316–325. IEEE Computer Society.

Zhang, K. (1995). Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition*, **28**(3), 463–474.

Zimmermann, T., Weissgerber, P., Diehl, S., and Zeller, A. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, **31**(6), 429–445.

Curriculum Vitae

Personal Information

Name	Beat Fluri
Nationality	Swiss
Date of Birth	July 16, 1979
Place of Birth	Basel, Switzerland

Education

2004 - 2008	Doctor in Informatics in the Software Engineering Group, Department of Informatics, University of Zurich, Switzerland Subject of dissertation: "Change Distilling – Enriching Software Evolution Analysis with Fine-Grained Source Code Change Histories" Advisor: Prof. Dr. Harald C. Gall External Examiner: Prof. Dr. David Notkin
2004	Master of Science ETH in Computer Science and dipl. Informatik-Ing. ETH at the Chair of Software Engineering, Department of Computer Science, Swiss Federal Institute of Technology (ETH) Zurich, Switzerland Subject of Master thesis: "Reflection for Eiffel" Advisor: Prof. Dr. Bertrand Meyer
1999 - 2004	Student of Computer Science at the Swiss Federal Institute of Technology (ETH) Zurich, Switzerland Minor in Business Administration
1990 - 1998	Gymnasium Kirschgarten in Basel, Switzerland
1986 - 1990	Primary School in Basel, Switzerland

Change Distilling

Enriching Software Evolution Analysis with Fine-Grained Source Code Change Histories

Software systems have to evolve over their life-cycle or they become progressively less useful. The reasons of why software is continuously changed are manifold: Features are added or adapted because of changing requirements; bugs have to be fixed because of faults in the software; or the software has to be migrated because of modernization. One negative effect of the continuing change is the software aging phenomenon. As software is changed from people unaware of the initial design concepts and, mostly, under time-pressure software becomes larger, more complex, and less understandable. As a result, in the last decade, several techniques have been developed to understand the negative impact of continuing change by analyzing change in general and source code change in particular.

The approaches developed so far suffer from the coarse-grained information available for changes. They rely on data provided by versioning systems, which keep track of changes by storing the text differences of a particular file. Changes at the level of source code entities are not considered. In addition, a precise definition and a classification of source code changes are still missing. Both are key to extract and analyze source code changes, and eventually understand the negative impact of continuing change. We therefore claim: Extracting, classifying, and analyzing fine-grained source code changes from the history of software systems provide useful insights into problems of continuing change and can identify support mechanisms to reduce them.

The key contribution of this dissertation is change distilling, a methodology to define, classify, extract, and analyze fine-grained source code changes. Change distilling provides a taxonomy of source code changes which defines source code change types according to tree edit operations in the abstract syntax tree. Our change distilling algorithm applies tree differencing pairwise on subsequent versions of abstract syntax trees to extract the tree edit operations.

We provide three empirical experiments to show the benefits of extracting fine-grained source code change types. First, we analyze the source code and comment co-change behavior in the evolution of eight software systems. We show that in cases where comments are adapted to source code changes, the related changes happen in the same revision. We also show that in half of these software systems API comments are adapted several revisions after the source code change happened.

Second, we explore whether certain change types appear frequently together. For that we use hierarchical agglomerative clustering to discover change type patterns and present a catalogue of change type patterns. The results from a commercial software system show that certain control flow changes are due to source code cleanup activities, that exception flow is used differently in different system parts, and that API convention changes are spread over many releases.

Third, we investigate whether methods exist whose invocations are significantly more affected by context and update changes than other methods, and whether we can reveal change patterns among these invocation changes. We develop an approach that ranks how often context and update changes were applied to invocations of a particular method and whether these changes were bug fixes. In addition, we extract patterns of context and update changes to assess whether they can be used to provide valuable change suggestions.

The results of our three software evolution experiments provide enough evidence that the analysis of change types helps in understanding software evolution and provides means to support developers in their daily work.